Practical fine-grained static slicing of optimized code

Michael D. Ernst
mernst@research.microsoft.com

July 26, 1994

Technical Report
MSR-TR-94-14

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Practical fine-grained static slicing of optimized code

Michael D. Ernst
mernst@research.microsoft.com

July 26, 1994

Abstract

Program slicing is a technique for visualizing dependences and restricting attention to just the components of a program relevant to evaluation of certain expressions. Backward slicing reveals which other parts of the program the expressions' meaning depends on, while forward slicing determines which parts of the program depend on their meaning. Slicing helps programmers understand program structure, which aids program understanding, maintenance, testing, and debugging; slicing can also assist parallelization, integration and comparison of program versions, and other tasks.

This paper improves previous techniques for static slicing. Our algorithm is expression-oriented rather than based on statements and variables, resulting in smaller slices. A user can slice on any value computed by the program—including ones that are not, or cannot be, assigned to variables. The slicer accounts for function calls, pointers, and aggregate structures. It takes advantage of compiler analyses and transformations, resulting in more precise slices, and bypasses syntactic constraints by directly constructing executable slices. These techniques are implemented in a slicer for the C programming language that accepts input via mouse clicks; operates on the value dependence graph, a dataflow-like program representation that is especially well-suited to slicing; and displays closure slices by highlighting portions of the program in the programmer's editor.

1 Introduction

A program slice [Wei84] captures a subset of a program's behavior. A slice represents either all parts of a program that can affect the *slicing criterion*—the expression of interest to the user—or all parts of the program whose run-time values may depend on the slicing criterion's value. Slices are usually represented to the user as a subset of the original program. In programming environments, slices demonstrate the relationships between different parts of a program; in program manipulation, slices limit how much of the program is manipulated. We present a slicer that improves three aspects of previous slicers: the size of the slices produced, programmer control over the slicing criterion, and support for a full, practical language.

First, our slicer produces more precise slices, restricting attention to a smaller subset of the program. The slicer decreases slice size by exploiting compiler analyses and transformations, thereby removing false dependences, eliminating dead code, and otherwise improving the resulting slices. Representing the program in terms of the original source requires maintaining a correspondence between the optimized and unoptimized versions of the program. The slicer also decreases slice size by constructing slices out of expressions rather than full statements. In a language that permits expressions to have side effects, only the relevant portion of each statement is included. Executable slicing can generate code directly from such slices without regard to syntactic constraints, since the slice is not conflated with its representation in terms of the original program.

Second, our slicer enhances programmer control over slicing: instead of restricting slicing criteria to variable references, a slicing criterion may specify any value computed by the program—including intermediate results, structure components, and non-first-class values such as I/O streams and the stack.

Richer and more precise specification of slicing criteria results in finer, more intelligible slices. We also discuss techniques, such as depth-limiting, for narrowing slices and making them more intelligible.

Third, we slice a real programming language. We enhance work on slicing in the presence of pointers and function calls. We introduce techniques for handling arrays, structures, unions, and expressions with side effects. Our slicer supports the entire C programming language (except `longjmp`). The implementation is written in Scheme [Han93] and integrated with GNU Emacs [Sta94] for input and output; it uses the value dependence graph [WCES94] as its intermediate representation.

1.1 Types of slices

Venkatesh [Ven91] categorizes slices into eight types according to three distinctions: backward vs. forward, executable vs. closure, and static vs. dynamic. This classification will help us to situate this research with respect to previous work.

A *backward slice* represents those program components affecting the values produced by execution of a set of expressions (the slicing criterion). A *forward slice* specifies all components of the program that may be affected by the slicing criterion values. For concreteness, all further references are to backward slicing, but our techniques are equally applicable to forward slicing.

Execution of code generated from an *executable slice* produces the same values for the slicing criterion as an execution of the original program on the same inputs. The traditional approach to executable slicing has been to produce a *compilable slice*, a syntactically correct program subset, then to compile it. Syntactic constraints complicate the construction of compilable slices and force the inclusion of extraneous program components. A *closure slice* (named for the graph reachability algorithm that computes it) contains only the components related to the slicing criterion and is more useful to programmers in understanding code behavior. We use a single slice for both types of slicing, either generating code directly from it or using it to direct highlighting of a program.

A *static slice* includes the program components that *may* have an effect on (or may be affected by) the slicing criterion values. A *dynamic slice* gives definite information for some particular execution by maintaining an execution trace of a running program. This paper discusses only static slicing, which uses analysis to discover dependences.

2 Slicing the value dependence graph

Our slicing algorithm operates on the value dependence graph (VDG) [WCES94], a sparse, parallel, functional, dataflow-like program representation. The VDG is composed of nodes which represent computation and arcs which carry values between computations. Figure 1 shows a program and its VDG representation.

The basic slicing algorithm [OO84] on the VDG is extremely simple: the slice consists of all computations encountered in a graph traversal starting at the slicing criterion. A backward slice traversal follows consumer-producer arcs; at call results, it proceeds to all corresponding call returns, and at formal parameters it proceeds to corresponding actual parameters. Because each VDG node is visited at most once, this algorithm runs in time linear in the size of the slice, which is no larger than the VDG. Figure 2 shows a slice of the procedure of figure 1.

The VDG is an attractive representation for slicing; the advantages of our slicing approach accrue both from the VDG's features and their exploitation by our algorithms. Because the VDG's unit of granularity is the computation rather than the statement, slicing criteria and results are finer. Because the graph directly links value producers with consumers, all dependences are made explicit in a single graph. Because all values and computations, including those on the global store and I/O streams, are explicitly represented, the algorithm need not account for hidden constraints and effects.

```
void sum_product(int n)
{
  int sum = 0;
  int product = 0;
  int i = 0;

  while (i++ < n)
    { sum = sum + i;
      product = product * i;
    }
  printf("Sum %d, product %d",
         sum, product);
}
```

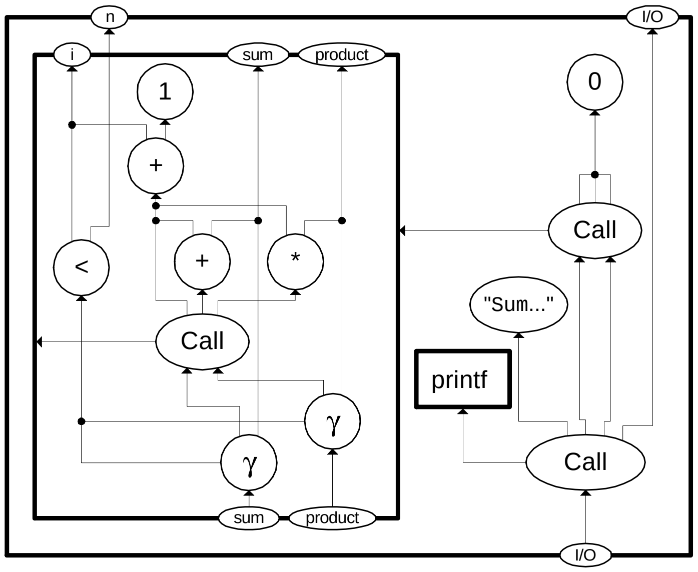


Figure 1: A procedure and its VDG [WCES94] representation. The arrows point from consumers to producers of values. The γ node is a selector whose output is its left or right input, depending on whether its boolean argument (attached to the side of the γ node) is true or false. All looping and other control flow is represented by function calls; the heavy boxes represent function bodies, with arguments along the top edge and return values along the bottom edge.

The basic slicing algorithm is simple, fast, and easy to understand, but imprecise: the slices it produces are larger than necessary. We will improve this algorithm by extending it to address pointers and aggregate values (section 4) and procedure calling context (section 5). Other sections discuss our techniques for expression-oriented slicing (section 3), executable slicing (section 6), and slicing optimized code (section 7).

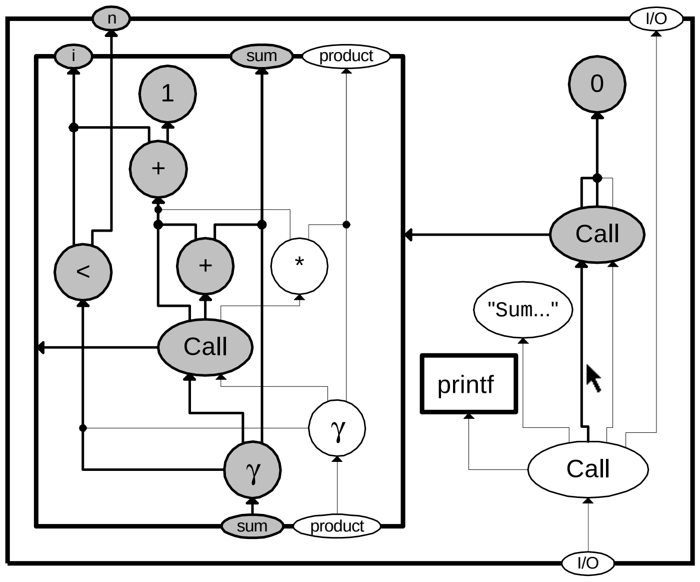
3 Expression-oriented slicing

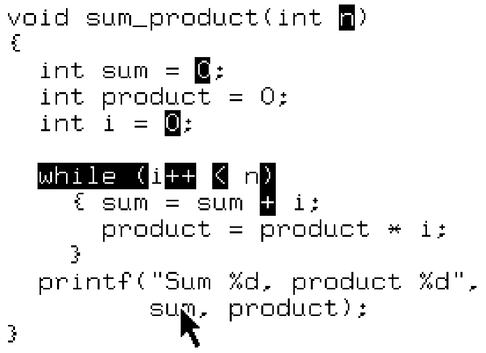
Our slicing algorithm works at the expression level for input, computation, and output. Slicing criteria can represent any value computed by the program, including those which are not, or cannot be, named by variables. Slices, which are defined in terms of the VDG rather than a textual representation, can include partial statements. Finally, our closure slices accurately convey information to the user and our executable slices (discussed in section 6) do not include extraneous computations.

3.1 Slicing criteria

A traditional slicing criterion is a statement and a set of variables it references. We extend slicing criteria to include any expression in the program (including those computing values undenotable in the programming language) and variable references which do not appear explicitly in an expression or statement. The slicing criterion always represents a single value rather than a whole computation, which may compute multiple values.

Allowing intermediate results (subexpressions) as slicing criteria permits users to slice on components of a computation that are not assigned to separate variables. This feature enables programmers to restrict attention to the parts of statements or expressions. Values not assigned to variables are more likely to appear in code using complicated coding constructs, for which slicing is especially needed to untangle the computation. Examples from real programs include components of aggregate values, values





```
void sum_product(int n)
{
  int sum = 0;
  int product = 0;
  int i = 0;

  while (i++ < n)
    { sum = sum + i;
      product = product * i;
    }
  printf("Sum %d, product %d",
         sum, product);
}
```

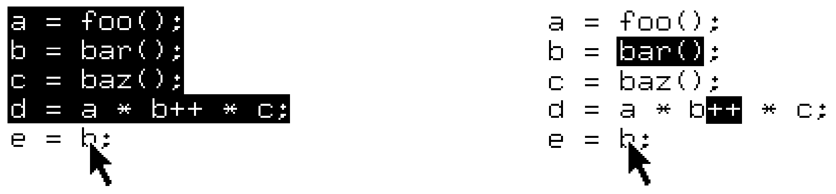
Figure 2: Slicing the VDG of figure 1 with slicing criterion `sum` in the call to `printf`. The slice indicates which computations can affect the slicing criterion's run-time value. In the VDG, the edges and nodes in the slice are shown thicker and shaded, respectively, and the slicing criterion is not a whole node but the indicated edge (value). Given the VDG slice, communicating it to the user is a user interface issue. The screen dump on the right shows how our system highlights, in the programmer's editor, the computations contained in the slice; see section 3.3 for discussion.

addressed indirectly through pointers, expressions that use values computed earlier in the expression's evaluation, very large expressions, and expressions that set, then overwrite, a variable. Rewriting code to accommodate variable-based slicing criteria is an inconvenient (sometimes impossible) and error-prone alternative to slicing on arbitrary expressions.

Values that cannot be assigned to variables are inexpressible in terms of variable references. For determining which computations can affect the output produced by a program up to a particular point, the slicing criterion is the output stream, which is not denotable in most programming languages. Similarly, slicing on the state of the heap or stack reveals where allocations and deallocations occur.

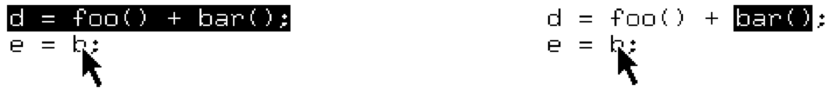
Our slicer supports these queries because all computations are explicitly represented in the VDG. Slicing on an expression that produces multiple values produces a menu of the expression's results, one of which is selected as the slicing criterion. This mechanism also permits slicing on variables that are not referenced at a program point, which is not supported by other slicers (except [Wei84]). Since the global store is represented explicitly in the graph, any live component of it can be sliced upon.

There are two complications to slicing on arbitrary variable references. First, optimization (see section 7) can reorder, eliminate, duplicate, and coalesce computations. In serialized code, zero, one, or more locations in the intermediate representation may correspond to a source location, and frequently there will be no point at which all computations preceding a program point, but no computations following it, have been performed. Second, the VDG routes values only where they are demanded: in figure 1, the I/O stream is not passed into the loop, and the global store has been eliminated entirely. As a result, not all program values are slicable at a particular expression, though they could be in a serialized VDG which threaded all values through every computation. Any expression appearing in the program can be used in a slicing criterion, however, and any value computed by the expression is slicable.



```
a = foo();                  a = foo();
b = bar();                  b = bar();
c = baz();                  c = baz();
d = a * b++ * c;            d = a * b++ * c;
e = b;                      e = b;
```

Figure 3: Comparison of statement-oriented and expression-oriented slices for slicing criterion `b` in `e = b;`. The call to `baz` is assumed not to set `b`.



```
d = foo() + bar();          d = foo() + bar();
e = b;                      e = b;
```

Figure 4: Comparison of statement-oriented and expression-oriented display of slices. In the program from which this fragment is taken, `bar` modifies `b`, but `foo` does not. For slicing criterion `b` in `e = b;`, the slice includes the call to `bar` but not the call to `foo`. Portions of the body of `bar` (not shown) are also highlighted.

3.2 Computing slices

When constructing a slice, our slicer includes individual computations rather than whole statements. If expressions can have side effects or statements can have multiple effects, this finer-grained approach, which uses the computation's true granularity rather than that of the programming language, improves slice precision.

Figure 3 compares statement-oriented and expression-oriented slices for slicing criterion `b` in `e = b;`. The statement-oriented slice includes `d = a * b++ * c;` because it modifies `b`; and because that statement is included, all the computations it depends on are also included, even though the computations of `a` and `c` are irrelevant to the final value of `b`. The expression-oriented slice effectively indicates that the final value of `b` is one greater than the value of the call to `bar`.

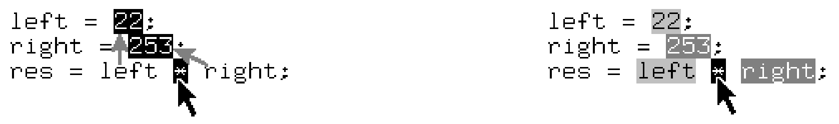
We define a slice in terms of the intermediate representation on which the slicing algorithm operates. Unlike historical definitions that require a slice to be a set of statements or a syntactically correct program, our definition does not conflate the slice with its on-screen representation. We do not try to make a single textual representation serve the different purposes of closure slicing (to provide a programmer with information about dependences and value flow) and executable slicing (to generate an executable whose behavior subsets that of the original program).

3.3 Displaying closure slices

Given a slice—a VDG subset—how to display it is a user interface issue. Expression-oriented slices can be displayed in a traditional statement-oriented fashion, but doing so nullifies some of the advantage of slicing at a finer granularity. By default we highlight only the expressions corresponding to computations in the slice, as shown in figure 4. The abbreviated output of expression slicing can be a boon, because it directs attention precisely to the relevant computation and to the flow of values in the program rather than making the user reconstruct that information. When the goal is to aid a programmer, there is little motivation to include full statements or to make the slice's representation syntactic, so long as the slice is represented in the context of the full program.

Static slices are frequently criticized for being too large and difficult to understand. Interesting results tend to depend directly or indirectly on most of the previous computation, so a full static slice includes most of a program. Determining relationships among the many included components of a slice, requires the programmer to manually repeat the slicer's job.

Our implementation uses *depth-limited slices* to show immediate dependences at the programmer's



```
left = 22;
right = 253;
res = left * right;
```

```
left = 22;
right = 253;
res = left * right;
```

Figure 5: Methods for making dependences explicit in large slices. On the left, arrows link consumers with producers; on the right, colors show those relationships. We use a different technique, depth-limited slicing, to provide this information to the programmer on demand.

request. An unlimited slice includes all computations the slicing criterion depends on, anywhere in the program. A depth limit of d curtails the graph traversal algorithm to pass through only d computations (selectors (`ifs`) and weak updates are treated specially). A depth-0 slice a value's definition points by including only the computation's operator. A depth-1 slice includes the operator and (the operators of) its operands. The slices in figure 5 have depth 1.

Figure 5 demonstrates two alternate techniques for explicitly showing all dependences in a slice: drawing arrows and using colors. These techniques can be distracting, however, and usually provide more information than desired.

4 Slicing with pointers and aggregate values

In order to effectively slice code containing pointer operations, our slicing algorithm uses a points-to analysis [HEGV93, EGH93], which indicates the possible values for each location-valued construct in the program.

Variable references and pointer dereferences are represented in the VDG by lookup nodes, which take two arguments—a location and a store—and return the contents of the location in the store. Update nodes take three arguments—a location, a store, and a value—and return a modified store in which the location has been set to the value. For ordinary variable references and assignments, the location is a constant, and in the absence of arbitrary pointer manipulation, VDG construction and optimization cancel out many update-lookup pairs, improving the graph's sparseness.

When the slicing traversal encounters a lookup node, its location argument is sliced in the usual way. The slice does not include the lookup node's entire store argument, however, because not all operations that contribute to the store are relevant to the lookup node's value. Instead, the store is treated as an aggregate value, each component of which is individually included in (or excluded from) a slice. The slice traversal proceeds from the store along all the locations that may be looked up—that is, along all of the locations specified by the points-to analysis for the lookup node's location argument.

We can think of slicing along a store value as occurring individually for each location l in the store. Each update node encountered falls into one of three categories:

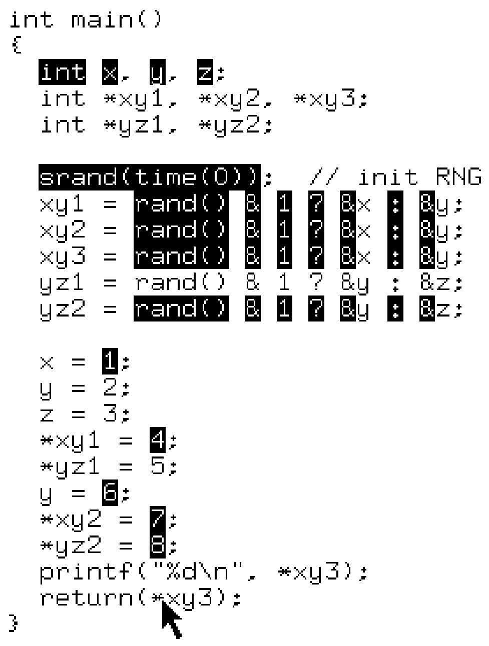
strong update of l: the update node's location argument always returns l. The slice traversal proceeds to the update node's value and location inputs, but not to its store input.

weak update of l: the update node's location argument may return l. The slice traversal proceeds to the update node's value and location inputs, and also proceeds along its store input, with respect to l.

other: the update node's location argument never returns l. The slice traversal proceeds along the update node's store input only.

The actual algorithm considers all locations simultaneously and takes advantage of pointer equality of unknown pointers when possible. Figure 6 shows the result of slicing a program containing pointer manipulations.

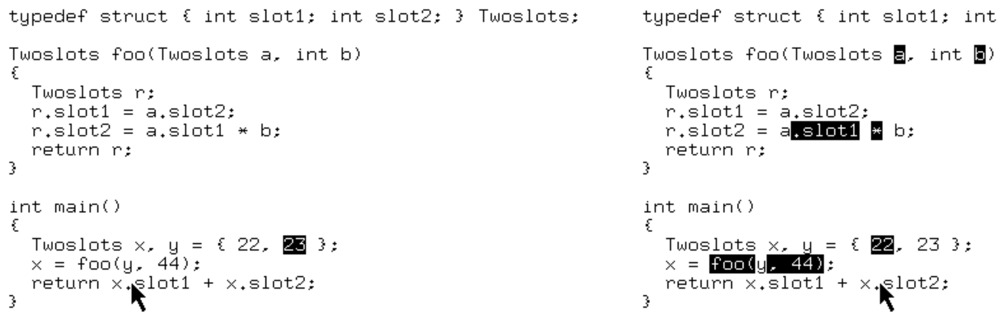
The slicing algorithm treats other aggregate values just like the store (see figure 7 for an example). Ignoring for the moment VDG nodes with multiple corresponding source texts, a scalar value edge is



```
int main()
{
  int x, y, z;
  int *xy1, *xy2, *xy3;
  int *yz1, *yz2;

  srand(time(0));  // init RNG
  xy1 = rand() & 1 ? &x : &y;
  xy2 = rand() & 1 ? &x : &y;
  xy3 = rand() & 1 ? &x : &y;
  yz1 = rand() & 1 ? &y : &z;
  yz2 = rand() & 1 ? &y : &z;

  x = 1;
  y = 2;
  z = 3;
  *xy1 = 4;
  *yz1 = 5;
  y = 6;
  *xy2 = 7;
  *yz2 = 8;
  printf("%d\n", *xy3);
  return(*xy3);
}
```

Figure 6: Slicing in the presence of pointer manipulation. Pointer **xy***n* points to either **x** or **y**, and **yz***n* points to **y** or **z**. The final value pointed to by ***xy3** is 1, 4, 6, 7, or 8, depending on the values of four of the calls to **rand**, the pseudorandom number generator. The highlighting of **int x, y, z** indicates that the addresses of those variables are manipulated.

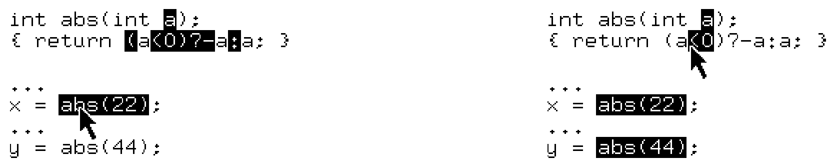
traversed only once, and its inclusion in the slice implies inclusion of its producer (and its producer's input values). When an aggregate value edge is traversed, the demanded components are noted, and the traversal proceeds at the corresponding inputs of the aggregate-producing node. An edge may be traversed a number of times equal to the number of components of the value on the edge. The algorithm runs in time linear in the number of values computed by the program instead of the number of nodes in its VDG. In the worst case, this could raise the cost of the slicing algorithm from linear to quadratic in the size of the program, but in practice it only adds a small constant factor. The size of the representation is not increased (beyond the storage requirements of the points-to analysis).



```
typedef struct { int slot1; int slot2; } Twoslots;

Twoslots foo(Twoslots a, int b)
{
  Twoslots r;
  r.slot1 = a.slot2;
  r.slot2 = a.slot1 * b;
  return r;
}

int main()
{
  Twoslots x, y = { 22, 23 };
  x = foo(y, 44);
  return x.slot1 + x.slot2;
}
```

```
typedef struct { int slot1; int slot2; } Twoslots;

Twoslots foo(Twoslots a, int b)
{
  Twoslots r;
  r.slot1 = a.slot2;
  r.slot2 = a.slot1 * b;
  return r;
}

int main()
{
  Twoslots x, y = { 22, 23 };
  x = foo(y, 44);
  return x.slot1 + x.slot2;
}
```

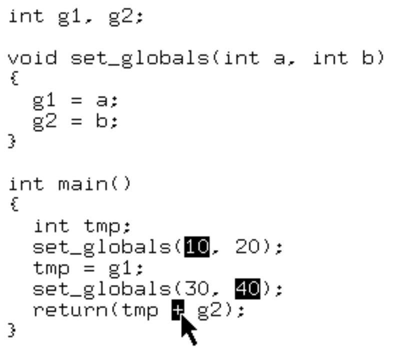
Figure 7: The first slot of **foo**'s return value depends directly on the second slot of formal **a**, and the second slot of the result value depends indirectly on the first slot of formal **a** and on formal **b**. Slices on **x.slot1** and **x.slot2** are shown, with constant folding turned off. The traditional method includes parts of **foo** in a slice on **x.slot1**; we omit it because there is no dependence on its body.



```
int abs(int a);
{ return (a<0)?-a:a; }

...
x = abs(22);
...
y = abs(44);
```

```
int abs(int a);
{ return (a<0)?-a:a; }

...
x = abs(22);
...
y = abs(44);
```

Figure 8: An interprocedural slicer must account for calling context to produce precise slices. When slicing on the one call to `abs` (as on the left) a naive interprocedural slicer would enter the procedure body, discover that the formal parameter is demanded, and then add to the slice the actuals of *all* calls to `abs`. A more accurate slice includes the actual parameter only for the call being sliced upon. On the other hand, a slice that starts inside the function body (as on the right) includes both of the calls to the function, since either of those actual parameters can affect the run-time value of an expression in the function body.



```
int g1, g2;

void set_globals(int a, int b)
{
  g1 = a;
  g2 = b;
}

int main()
{
  int tmp;
  set_globals(10, 20);
  tmp = g1;
  set_globals(30, 40);
  return(tmp + g2);
}
```

Figure 9: An accurate slice must be able to include different actual parameters at different call sites.

5 Interprocedural slicing

The basic slicing algorithm of section 2 handles interprocedural slicing by following, during the graph traversal, dependence links from formal parameters to actual parameters and from call results to formal return values. That algorithm constructs unnecessarily large slices, because it does not account for calling context. Not all formal-to-actual links should be followed, as illustrated by figure 8.

The obvious technique is to process each procedure body anew for each call, propagating the results to just that call site [HDC88]. However, this method does not run in time proportional to the size of the program, and recursion complicates guaranteeing its termination and correctness.

Our solution to this problem is patterned after Horwitz's [HRB90], but we produce finer slices by not including entire calls when only some actuals are demanded, by including only the relevant parts of aggregate values, and by omitting entire procedure bodies where appropriate. Our algorithm also permits different call sites to include different actual parameters, as shown in figure 9. Before slicing, a summary of the dependences of each procedure return on each formal parameter is computed. Slicing then proceeds in two stages: one for procedures that may appear in the call chain when the slicing criterion value is computed, and one for other procedures.

5.1 Summary dependence information

When a slice traversal encounters a procedure call, the slice must include appropriate parts of the procedure body, and it must proceed at the demanded actual parameters of the call. The latter goal does not require entering the procedure body, if a summary of dependences of procedure return values on formal parameters (and thus of call results on actual parameters) has been computed ahead of time.

The summary indicates which formal parameters each return value depends on. Additionally, each dependence is marked as direct or indirect. A direct dependence indicates that the formal is returned

Definitions:

$$\begin{aligned}
\textsc{summary}(ret) &= \text{the formals of return node } ret\text{'s procedure on which } ret \text{ depends}\\
\textsc{formals}(v) &= \text{the formals of value } v\text{'s enclosing procedure on which } v \text{ depends}\\
in(n) &= \text{the inputs of node } n\\
out(n) &= \text{the outputs of node } n
\end{aligned}$$

Dataflow equations for node n:

return node: $\textsc{summary}(n) = \textsc{formals}(in(n))$

formal node: $\textsc{formals}(out(n)) = \{n\}$

```
call node:  let $p$ = the called procedure
            for each $res \in out(n)$
                let $ret$ = the formal return in $p$ corresponding to $res$
                let $f \rightarrow a$ map from formals of $p$ to actuals of $n$
                $\textsc{formals}(res) = \bigcup_{x \in \textsc{summary}(ret)} \textsc{formals}(f \rightarrow a(x))$
```

otherwise: $\textsc{formals}(out(n)) = \bigcup_{i \in in(n)} \textsc{formals}(i)$

Figure 10: Dataflow equations for computing summary dependences of returns on formal parameters, giving SUMMARY(ret) for each return node ret. The equations used by our algorithm also address aggregate values, multiple callees at a call site, free values of procedures, and nontrivial correspondences with the source program due to optimization.

unchanged; an indirect dependence indicates that the formal may participate in computation to determine the return value. When a return value depends directly on a formal, a slice that includes a corresponding call result need not include any of the procedure body, because there is no dependence on the body. Figure 7 gives an example of dependences on formals; figures 7 and 9 show exclusion of procedure bodies from a slice due to direct dependences.

We compute the formal dependence summary by finding a fixed point solution for the set of forward flow equations shown in figure 10, which can be solved using an iterative worklist algorithm. For arbitrary call graphs, this information cannot be computed in a single pass.

5.2 Slicing algorithm

After formal dependences have been computed, the interprocedural slicing algorithm proceeds in two stages. The first stage addresses procedures that may be in the current call chain whenever a slicing criterion value is computed at run-time. When formals are encountered, the graph traversal ascends to the corresponding actuals at *all* call sites. When calls are encountered, the slice continues at the formals indicated by the formal dependence analysis. The procedure bodies are not entered, but the demanded results are noted on a worklist.

The second stage processes the worklist, slicing the bodies of functions whose return values can affect the slicing criterion. Since the slice has already proceeded at the appropriate actual arguments, no more work is needed when their formals are encountered. Procedure calls are skipped over and the demanded return values added to the worklist, as before. This stage proceeds until the worklist is empty; each

```
procedure backward-slice(slicing-criterion)
  let ascending-worklist = slicing-criterion
  let nonascending-worklist = {}
  let slice = {}

  for each node in ascending-worklist
    if node is in slice
      done
    else add node to slice
      if node is a formal
        add all corresponding actuals to ascending-worklist
      else if node is a call
          add all associated returns to nonascending-worklist
          add call actuals to ascending-worklist (according to summary dependence)
        else add all dependent nodes to ascending-worklist

  for each node in nonascending-worklist
    if node is in slice
      done
    else add node to slice
      if node is a formal
        done
      else if node is a call
          add all associated returns to nonascending-worklist
          add call actuals to nonascending-worklist (according to summary dependence)
        else add all dependent nodes to nonascending-worklist

  return slice
```

Figure 11: Pseudocode for the interprocedural backward slicing algorithm. For clarity, it ignores the subtleties of depth limiting, direct summary dependences, source correspondences, aggregate values, and so forth.

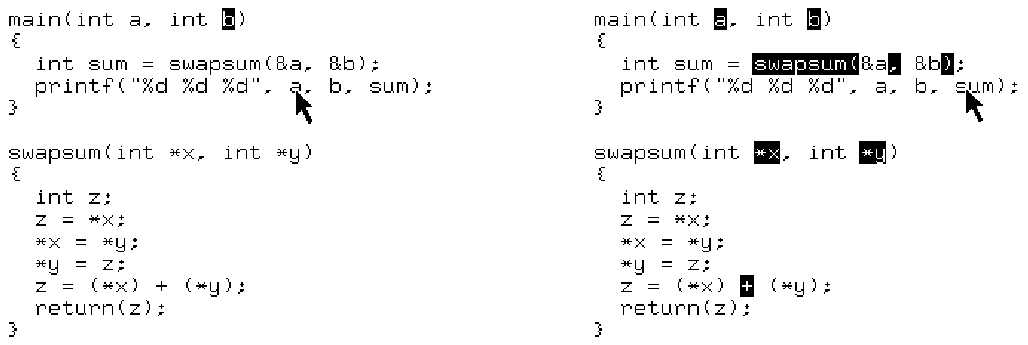
procedure return value is processed at most once (by either stage).

Figure 11 gives pseudocode for the interprocedural slicing algorithm. The complexity of this algorithm is linear in the number of values computed by the (unoptimized) program; the naive slicing algorithm which does not account for calling context has the same complexity.

6 Executable slicing

Executable slicing transforms one or more programs into object code, either by eliminating parts of a program or by combining multiple programs. When used for testing [LW87, KSF90, PC90, FSKG92], parallelization [Wei83, BW88], or program decomposition [RY88, Bin93], the goal is a smaller, faster program that computes the same slicing criterion values as the original program, but without computing other, irrelevant results. When used for integration of compatible versions of a program [RY89, HPR89, HR90, HRB90, Bin91, Bal93], an executable slice combines all the features of the various versions. A closure slice, on the other hand, helps a programmer visualize dependences without regard to syntactic constraints.

A single type of slice, consisting of VDG nodes and sources, can serve both closure and executable slicing, and can produce smaller, faster object code than compilable slices (traditional executable slices). For closure slicing, the slice directs highlighting of the program, and for executable slicing, code is



```
main(int a, int b)
{
  int sum = swapsum(&a, &b);
  printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
  int z;
  z = *x;
  *x = *y;
  *y = z;
  z = (*x) + (*y);
  return(z);
}
```

```
main(int a, int b)
{
  int sum = swapsum(&a, &b);
  printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
  int z;
  z = *x;
  *x = *y;
  *y = z;
  z = (*x) + (*y);
  return(z);
}
```

Figure 12: Interprocedural slicing with pointers. Jiang's implementation [JZR91, §3.2.2] of Horwitz's algorithm [HRB90] includes the entire program when slicing on `a` or `sum`; our slices are shown above.

generated directly from the slice.

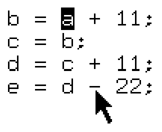
A compiler that uses the VDG as its intermediate representation easily generates code from a slice, because there is no real difference between a slice (a subset of a VDG) and a full VDG. Integrating the slicer with a compiler obviates the need to write a separate front end. The slicer can take advantage of the compiler's analyses and transformations, leading to significantly better dependence information than available from the raw program text (see section 7). Existing back end mechanisms can deal with dead code, arity raising and lowering, procedure specialization, and so forth, and the slice provides the back end precise liveness information.

Our executable slices preserve only program behavior related to the slicing criterion. Computations in the original program that loop forever or raise an exception are not included in the slice unless they contribute to the computation of the slicing criterion values. Therefore, the executable slice may terminate or produce an answer even when the original program would not have. On the other hand, if the executable slice fails to terminate or terminates exceptionally, then the original program would have done the same. The same conditions apply to optimized programs in general.

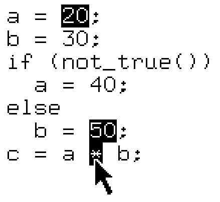
6.1 Compilable slicing

The traditional approach to executable slicing has been to produce a *compilable slice*, which is a syntactically correct program subset, then to compile it. Syntactic constraints complicate the construction of compilable slices and force the inclusion of extraneous program components [Bin93]. For instance, an compilable slice unnecessarily includes the entire program for figures 3, 4, 5, 7, 9, 12, and 13 (left side).

Compilable slicing spends much effort determining which extra portions of the program (besides the truly demanded computations) to include in order to make the resulting text syntactically and semantically correct. If the slice contains a particular parameter in any call of a function, then every call in the slice must include that parameter. If an expression's side effects force its inclusion in a slice, then any containing expressions and statements must appear in the slice. If a value is named by a particular variable at a statement in the slice, then the statement assigning the value to that variable must be included in the slice. Including the proper control-flow statements (such as `goto`) requires special care [JZR91, BH92, Bal93, CF93, Agr94]. Only a limited class of optimizations can be handled [Tip94], since optimized programs can be inexpressible in the original programming language. An optimizing compiler cannot remove all of the extra constructs added by compilable slicing without knowledge of which constructs are essential (part of the slice) and which are incidental (added for the sake of compilability) and without extra analyses that may be more complicated than slicing itself.



```
b = a + 11;
c = b;
d = c + 11;
e = d - 22;
```



```
a = 20;
b = 30;
if (not_true())
  a = 40;
else
  b = 50;
c = a * b;
```

Figure 13: Slices on unoptimized code include all the text in these code fragments. On the left, an expression-oriented slice of optimized code shows that the subtraction's value is the same as that of `a`. On the right, if analysis can determine that the call to `not_true` returns false, the slice can show the actual arguments to the multiplication. (The right-hand screen dump was generated with constant folding disabled; with constant folding enabled, the user is informed, "The value is 1000." A history mechanism can indicate how this fact was determined.)

7 Slicing optimized code

The quality of a slice depends directly on the accuracy of the dependences discovered in the program. The original program, however, is a poor guide to the dependences inherent in its computation, since the source code may contain false dependences, dead code, and nonoptimal computations. Slicing optimized (analyzed and transformed) code eliminates these problems—but the results should be communicated to the user in terms of the original program text.

Compiler analyses discover information about the program, from resolving possibly aliased pointers to determining run-time values. Compiler transformations (such as constant folding, dead code elimination, and elimination of partial redundancies) update the program structure to reflect knowledge gleaned from analyses. Analysis and transformation remove and simplify dependence relations, making them a better approximation to the true dependences of the underlying computations, so slicing after optimization produces more precise and informative slices. Figure 13 demonstrates some advantages of slicing optimized code.

Optimization also improves liveness information. When a slicing criterion includes dead code, our implementation so indicates rather than showing the slice of that code with respect to the unoptimized program.

Slicing optimized code is especially powerful in the presence of abstractions such as high-level constructs and object-oriented features. Such abstractions insulate the programmer from low-level details, but manually tracing or verifying value flow through multiple layers of abstraction is tedious and error-prone.

Finally, the hardware usually runs optimized code, so the programmer must understand its behavior. The optimized code's behavior is consistent with that of the unoptimized version, though the optimized code better models the program's underlying computation.

7.1 Correspondences between source and optimized program

The most effective medium for specifying slicing criteria and displaying slices is the original program, which was used to specify the slicing criterion, with which the user may be familiar, and which contains comments, formatting, and context. However, the program text does not correspond directly to the optimized VDG.

We maintain, throughout the optimization process, a correspondence between the VDG and a source code graph. The two graphs are initially isomorphic, but transformations modify them so that the correspondence between their nodes becomes a many-to-many relation. A separate paper will describe the engineering challenges of maintaining this correspondence and manipulating the resulting graphs.

Maintaining the correspondence also enables a history mechanism to explain which transformations occurred and, therefore, why the slice may be different (often more precise) than the user expected. It also permits slices to be displayed as if no transformations occurred, even if the VDG has been optimized, when the naive view is desired.

Inclusion of a VDG node in a slice does not imply inclusion of all the corresponding sources. The VDG slice in figure 2 contains the constant zero, represented by a single VDG node, but when the slice is displayed, only the first and third '`0`' tokens are highlighted. We modify our slicing algorithm to visit each node at most once for each associated source text instead of visiting each node at most once. The slicing algorithm traverses the VDG and the source graph in concert, starting at the VDG and source nodes corresponding to each slicing criterion expression. This modification increases the algorithm's complexity from linear in the size of the VDG to linear in the size of the source code graph, which is approximately the size of the original program.

8 Related work

Weiser introduced the concept of slicing [Wei79, Wei84] and showed its application to debugging [Wei82, LW86, LW87] and parallelization [Wei83, BW88]. His slicing criteria consist of a statement and a set of variables not necessarily referenced in the statement, and his slices are syntactically correct subsets of the original program with the same termination behavior. Weiser treats slicing as a dataflow problem and provides an $O(ne \log e)$ algorithm which includes every call site of a function if any call is included.

Ottenstein and Ottenstein's linear-time slicing algorithm based on graph reachability [OO84] demonstrates the appropriateness of the program dependence graph (PDG) [FOW87, HPR88] for slicing. Slicing criteria are slightly restricted: variables must appear in the specified statement. Most subsequent research builds upon this work.

Horwitz et al. [HRB88, HRB90, Bin91, Bin93] extend the PDG algorithm to account for calling context. An interprocedural PDG (called the system dependence graph) summarizing transitive dependences due to calls is constructed from an attribute grammar that models procedure-call structure and the subordinate characteristic graphs of the grammar's nonterminals. Once the graph is constructed, slicing takes linear time (but aliasing degrades the performance, and Binkley's analysis [Bin93] assumes the program contains a constant number of global variables). Constructing a compilable slice requires specialization of each procedure for each subset of parameters demanded by a call site, so the result is not a subset of the original program. Livadas and Croll [LC92] build the system dependence graph incrementally, processing each procedure as it is encountered in a traversal. When a function call is encountered, processing of the current procedure is suspended and a partial solution is preserved; later refinements of that solution require reprocessing of callers. The algorithm processes contents of looping statements twice. Our interprocedural algorithm is similar, but constructs the summary by solving a set of dataflow equations and uses direct dependences to skip over procedure bodies which do not contribute any dependences.

Hwang et al. [HDC88] slice interprocedurally in the presence of self-recursion by computing a series of slices in which the recursion depth is limited: initially no recursive calls are followed, then one level, and so forth. The fixed point solution is achieved when increasing the level results in no larger a slice. The recursive and base cases of recursive procedures are assumed to be identifiable. Such slices are as accurate as those based on the system dependence graph but cannot be computed as efficiently [HRB90].

Jiang et al. [JZR91] handle pointers and arrays by means of an alias analysis [Wei80]. They introduce a dummy variable for each level of dereference of each pointer variable and a dummy literal for each variable whose address is taken. Modifications of the pointer count as modifications of these dummy variables, and pointer uses count as dummy uses. Other authors do not address general pointer manipulation, but Horwitz et al. [HRB90] give two techniques for dealing with aliasing introduced by call-by-reference function parameters. The first technique (also used by Livadas and Croll [LC92] to solve the same problem) converts the program into an alias-free one by duplicating each procedure for every possible alias pattern among its (by-reference) parameters and modifying formal parameters and call sites accordingly. This transformation increases the size of the program exponentially in the maximum number of parameters passed to a procedure—but since a global variable is a parameter to each procedure that might use or modify it, the blowup can be exponential in the size of the original program. The second technique of Horwitz et al. gives a less precise slice but has a lower cost: if two variables can be aliased and a definition of one can reach a use of the other, then a dependence is added between them. By contrast to these techniques, we use a points-to analysis [HEGV93, EGH93] and add support for aggregates such as C's `struct` and `union` types.

Ottenstein and Ottenstein [OO83, OO84] and Tip [Tip94] note that compiler transformations can be of use to a slicer but give no details. Unlike their schemes, our optimizations are unconstrained by the syntax of the original program.

Livadas and Croll [LC92] independently suggest slicing on a parse tree to improve statement-based slicers. Unlike our implementation, their algorithm includes irrelevant operations (such as the assignment to `sum` in figure 7 of their paper), and they communicate the slice to the user in terms of whole statements. Jackson and Rollins [JR94] note the drawbacks of using an entire PDG node as a slicing criterion: slicing criteria should stand for particular values, not whole computations, and demand on part of a computation does not imply demand on all its inputs. Their "chopping" technique, similar to the intersection of forward and backward slices, is the only technique besides ours to permit non-variables (such as the output stream) as slicing criteria; we also permit arbitrary expressions and arbitrary variable references, like Weiser [Wei84].

For further references to the extensive slicing literature, see Tip's survey [Tip94].

Acknowledgements

Daniel Weise, Ellen Spertus, Bjarne Steensgaard, Todd Knoblock, Roger Crew, and Erik Ruf provided stimulating discussions and helpful comments on earlier drafts of this paper. They also assisted in the creation of the programming environment which includes the slicer described in this paper.

References

[Agr94] Hiralal Agrawal. On slicing programs with jump statements. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, FL, June 20-24 1994.

[Bal93] Thomas Jaudon Ball. The use of control-flow and control dependence in software tools. Technical Report 1169, University of Wisconsin – Madison, August 1993. PhD thesis.

[BH92] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control flow. Technical Report 1128, University of Wisconsin – Madison, December 21, 1992.

[Bin91] David W. Binkley. Multi-procedure program integration. Technical Report 1038, University of Wisconsin – Madison, August 1991.

[Bin93] David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, March–December 1993.

[BW88] Lee Badger and Mark Weiser. Minimizing communication for synchronizing parallel dataflow programs. In *Proceedings of the 1988 International Conference on Parallel Processing, Volume II, Software*, pages 122–126, Penn State, August 1988.

[CF93] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of GOTO statements. Submitted for publication; previously titled "What is in a slice", October 19, 1993.

[EGH93] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. ACAPS Technical Memo 54, McGill Uiversity, School of Computer Science, November 1993.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FSKG92] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, 1(4):303–322, December 1992.

[Han93] Chris Hanson. *MIT Scheme Reference Manual.* MIT Scheme Team, Cambridge, MA, 1.41 beta for scheme release 7.3 edition, December 6, 1993.

[HDC88] J. C. Hwang, M. W. Du, and C. R. Chou. Finding program slices for recursive procedures. In *Proceedings COMPSAC 88: The Twelfth International Computer Software and Applications Conference*, Chicago, October 1988. IEEE Computer Society.

[HEGV93] Laurie J. Hendren, Maryam Emami, Rakesh Ghiya, and Clark Verbrugge. A practical context-sensitive interprocedural alias analysis framework for C compilers. ACAPS Technical Memo 72, McGill University School of Computer Science, Advanced Compilers, Architectures, and Parallel Systems Group, Montreal, Quebec, July 24, 1993.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[HR90] Susan Horwitz and Thomas Reps. Efficient comparison of program slices. Technical Report 982, University of Wisconsin – Madison, December 1990.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. Technical Report 756, University of Wisconsin – Madison, March 1988.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[JR94] Daniel Jackson and Eugene J. Rollins. Abstract program dependences for reverse engineering. Submitted to Second ACM SIGSOFT Symposium on Foundations of Software Engineering, June 1994.

[JZR91] Jingyue Jiang, Xiling Zhou, and David J. Robson. Program slicing for C — the problems in implementation. In *Proceedings, Conference on Software Maintenance 1991*, pages 182–190, Sorrento, Italy, October 15-17, 1991. IEEE Computer Society Press.

[KSF90] Miriam Kamkar, Nahid Shahmehri, and Peter Fritzson. Bug localization by algorithmic debugging and program slicing. In P. Deransart and J. Małuszyński, editors, *Proceedings, Programming Language Implementation and Logic Programming, International Workshop PLILP '90*, pages 60–74, Linköping, Sweden, August 20-22 1990. Springer-Verlag. LNCS 456.

[LC92] Panos E. Livadas and Stephen Croll. Program slicing. Technical Report SERC-TR-61-F, Computer and Information Sciences Department, University of Florida, Gainesville, FL, October 1992.

[LW86] Jim Lyle and Mark Weiser. Experiments on slicing-based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, *Proceedings of the First Workshop on Empirical Studies of Programmers*, pages 187–197, Washington, DC, June 5-6, 1986.

[LW87] James R. Lyle and Mark Weiser. Automatic program bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, Beijing, China, June 1987.

[MNB$^+$94] Lawrence Markosian, Philip Newcomb, Russell Brand, Scott Burson, and Ted Kitzmiller. Using an enabling technology to reengineer legacy systems. *Communications of the ACM*, 37(5):58–70, May 1994.

[NEK94] Jim Q. Ning, Andre Engberts, and W. (Voytek) Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

[OO83] Karl J. Ottenstein and Linda M. Ottenstein. High-level debugging assistance via optimizing compiler technology (extended abstract). In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-level Debugging*, pages 152–154, Pacific Grove, CA, March 20-23 1983.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 177–184, Pittsburgh, Pennsylvania, April 1984.

[PC90] Andy Podgurski and Lori A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[RY88] Thomas Reps and Wuu Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin – Madison, June 1988.

[RY89] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. In J. Díaz and F. Orejas, editors, *TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development. Vol.2: Advanced Seminar on Foundations of Innovative Software II and Colloquium on Current Issues in Programming Languages (CCIPL)*, number 352 in Lecture Notes in Computer Science, pages 360–374, Barcelona, Spain, March 13-17, 1989. Springer-Verlag.

[Sta94] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, Cambridge, MA, tenth edition, July 1994. ISBN 1-882114-03-5.

[Tip94] F. Tip. A survey of program slicing techniques. Report CS-R9438, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, Toronto, Ontario, Canada, June 26-28, 1991.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. Technical Report MSR-TR-94-03, Microsoft Research, Redmond, WA, April 13, 1994.

[Wei79] Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.

[Wei80] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, January 1980.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–52, July 1982.

[Wei83] Mark Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17(5):129–135, October 1983.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.