Empirical Study of Optimization Techniques for Massive Slicing

DAVID BINKLEY,
Loyola College in Maryland
MARK HARMAN,
King's College London
and
JENS KRINKE,
FernUniversität in Hagen

This paper presents results from a study of techniques that improve the performance of graph-based interprocedural slicing of the System Dependence Graph (SDG). This is useful in "massive slicing" where slices are required for many or all of the possible set of slicing criteria. Several different techniques are considered, including forming strongly connected components, topological sorting, and removing transitive edges.

Data collected from a test bed of just over 1,000,000 lines of code are presented. This data illustrates the impact on computation time of the techniques. Together, the best combination produces a 71% reduction in run-time (and a 64% reduction in memory usage). The complete set of techniques also illustrates the point at which faster computation is not viable due to prohibitive preprocessing costs.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments; E.1 [**Data Structures**]: Graphs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Slicing, Internal Representation, Performance Enhancement, Empirical Study

Authors' Addresses: David Binkley, Loyola College in Maryland, Baltimore, Maryland, 21210-2699, USA. binkley@cs.loyola.edu Mark Harman, King's College London, Strand, London, WC2R 2LS, UK. Mark.Harman@.kcl.ac.uk Jens Krinke, FernUniversität in Hagen, 58084 Hagen, Germany. Jens.Krinke@FernUni-Hagen.de

This research is supported by National Science Foundation grant CCR-0305330 and Engineering and Physical Sciences Research Council grants GR/R98938, GR/M58719 and GR/R71733. Copyright 2006 by David Binkley, Mark Harman, and Jens Krinke. All rights reserved.

1. INTRODUCTION

As originally introduced, program slicing is a source code extraction technique that allows a software engineer to extract an executable sub-program based upon a slicing criterion [Weiser 1981; 1982]. Program slicing has applications in debugging [Weiser 1982], testing [Binkley 1998; Harman and Danicic 1995; Hierons et al. 1999; Hierons et al. 2002], program comprehension [Binkley et al. 2000; Fox et al. 2001; Rilling et al. 2001; Zhao 2002], program decomposition [Gallagher and Lyle 1991] and integration [Binkley et al. 1995; Horwitz et al. 1989], software metrics [Bieman and Ott 1994; Ott and Thuss 1989; Longworth et al. 1986] and re- and reverse engineering [Canfora et al. 1994; Canfora et al. 1994; Cimitile et al. 1995a; 1995b].

In some applications of slicing, only several slices need be computed, but in others, particularly those where slicing is used as an intermediate step in the computation of dependence information, there is a need to slice on many or all of the possible slicing criteria. This "massive slicing" is the motivation for the present paper, which studies six dependence graph optimization techniques that support improved performance for massive slicing.

Ottenstein and Ottenstein [Ottenstein and Ottenstein 1984] first observed the suitability of the program dependence graph for computing (intraprocedural) slices as the solution to a reachability problem. The program dependence graph and its applications were also studied by Ferrante et al. [Ferrante et al. 1987]. Horwitz et al. later provided an algorithm for context-sensitive interprocedural slicing that computed slices as the solution to a graph reachability problem over the system dependence graph [Horwitz et al. 1990; Reps et al. 1994; Reps 1998]. It is this interprocedural formulation of the dependence graph which forms the subject of the studies in the present paper.

The context for the work reported here is that of providing assistance to tool designers making decisions regarding the incorporation of a particular graph technique and representation. The paper studies six techniques that improve the performance of computing context sensitive interprocedural slices as the solution to a graph reachability problem. Results collected from a test bed of just over 1,000,000 lines of source code quantify the impact on the running time (and thus the applicability) of the techniques. It is expected that these techniques may also be applicable to and beneficial for other related graph-reachability-based analyses. However, future work is necessary to formally verify this belief.

The remainder of the paper is organized as follows: Section 2 explains the applications of massive slicing (slicing on every possible slicing criterion). Section 3 provides background on program slicing and the tools used to collect the data. Section 4 describes the optimization techniques, while Section 5 reports the results of the study. Finally, Sections 6 and 7 present related work and then summarize the paper.

2. THE APPLICATIONS OF MASSIVE SLICING

This paper presents techniques for scaling up slicing so that it becomes possible to realistically slice medium to large programs on *every possible* slicing criterion. This section briefly describes some of the applications of this "massive slicing."

Following Weiser's original suggestion in his thesis [Weiser 1979], several authors have described techniques that use slicing to measure the cohesion of programs [Bieman and Ott 1994; Meyers and Binkley 2004; Longworth et al. 1986; Ott and Thuss 1989; Ott and Bieman 1992; Ott and Thuss 1993; Ott 1992]. In this approach, slices for all "principal variables" are constructed and the relationships between these slices are explored to provide a basis for cohesion measurement. For example, in a program where there is a large degree of overlap among the program's slices, this would tend to indicate that the program is cohesive; that is, there is a large degree of shared computation, upon which all the primary computational results depend. This observation is at the heart of the work on slice-based cohesion measurement.

In order to measure cohesion using this approach, it is necessary to construct a slice of all principal variables in the program. The definition of what constitutes a 'principal' variable is a parameter in these approaches to cohesion measurement. However, common choices [Ott and Thuss 1989; Ott and Bieman 1992; Ott and Thuss 1993; Ott 1992] include local variables whose value is output, global variables which are affected by a procedure, and a procedure's reference-value parameters. This approach can result in a large number of slices being required. For example, in a recent study of forward and backward slice sizes [Binkley and Harman 2005a] a code base of just over 1M lines of code was found to contain 1.8M global variable instances that would play the role of a principal variable according to this approach (*i.e.*, summed over all procedures, there are 1.8M global variable instances that would be deemed to be principal variables). Therefore, taking global variables alone, the cohesion measurement for this code base would require the construction of almost 2M slices.

Massive slicing is also required in applications of slicing to software maintenance, most notably in the technique of union slicing [Beszédes and Gyimóthy 2002; Danicic et al. 2004] and of decomposition slicing [Gallagher and Lyle 1991]. A union slice is constructed from the union of many slices from the program, each constructed for a different dynamic slicing criterion. The aim is to approximate the precise static slice. This cannot always be fully determined, because minimal slicing is known to be undecidable [Weiser 1979; 1984].

The goal of union slicing is to provide a lower bound on the value of the precise static slice, by combining the results of many dynamic slices. Together with the static slice itself (which provides an upper bound on the precise slice) this provides an approach to incrementally reducing the interval in which the precise slice can be determined to lie. Of course, to construct such a union slice requires that many slices be computed. The more slices that can be computed the greater the precision, so there is a direct connection between the precision of the approach and the level of massive slicing ability. While many different dynamic slicing algorithms exist, it is not uncommon for these algorithms to reuse the same dependence graph for all slices (for example, the algorithms of Agrawal and Horgan [Agrawal and Horgan 1990]).

In decomposition slicing, the goal is to construct a partial order of slices that decomposes the program under maintenance. From this decomposition, a set of complement slices can be constructed such that changes to the complement can be shown to leave certain decomposition slices unaffected. This approach supports maintenance activities that change the software; the decomposition and its complement delimit the ares of the software which can be safely changed without introducing ripple effects [Black 2001].

Other work on program analysis has used slicing as a bench mark application [Liang and Harrold 1999; Mock et al. 2002]. One aspect of this work was to assess the impact of different points-to analyses upon slice size. In order to ensure that the results were not biased by the choice of slicing criterion, the authors used as wide a selection of slicing criteria as possible [Liang and Harrold 1999]. The work of the present paper goes some way towards supporting this kind of investigation by increasing the chance that this form of study could routinely apply massive slicing, thereby removing possible experimenter bias in the choice of slicing criteria.

The authors developed the optimization techniques reported in the present paper in order to investigate, empirically, the nature of program dependence [Binkley and Harman 2004; 2005a; 2005b]. All of these studies required the construction of large numbers of slices, because slices had to be constructed for every valid slicing criterion in every procedure of every program. The ability to slice on every possible slicing criterion allowed for investigations of the relationship between forward and backward slicing [Binkley and Harman 2005a] and of the levels of predicate dependence, and of their correlation to parameter list size [Binkley and Harman 2004]. By a similar 'massive slicing' approach the authors were able to reveal the presence of dependence clusters, which may cause problems for maintenance [Binkley and Harman 2005b].

3. BACKGROUND

This section describes the interprocedural slicing algorithm, the subject programs, threats to the validity of the experiment, and the data collection environment.

3.1 Interprocedural Slicing of SDGs

The System Dependence Graph (SDG) is a collection of Procedure Dependence Graphs (PDGs) connected at call-sites by interprocedural control- and flow-dependence edges [Horwitz et al. 1990]. A PDG represents a procedure as a collection of vertices and edges. The vertices represent the components of the program (*e.g.*, assignment statements and predicates) and the edges represent dependences between them. With the exception of call statements, a single vertex represents predicates (*e.g.,* from `if` and `while` statements), assignment statements, etc. A call statement is represented using a *call* vertex and four kinds of *parameter vertices* that represent parameter passing: on the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, while in the called procedure it is represented by *formal-in* and *formal-out* vertices.

There are two kinds of edges in a PDG: control dependence edges and data dependence edges. A control dependence edge is labeled either **true** or **false**. Informally, it connects a predicate vertex v to a vertex u if, during execution, when the predicate represented by v is evaluated and its value matches the edge's label, then the program component represented by u will eventually be executed (provided the program terminates normally) [Horwitz et al. 1990; Ferrante et al. 1987]. The only kind of data-dependence edge used is the *flow dependence edge*. A

```
while (i < 10)
{
  sum += a[i]
  i = i + 1
{
```

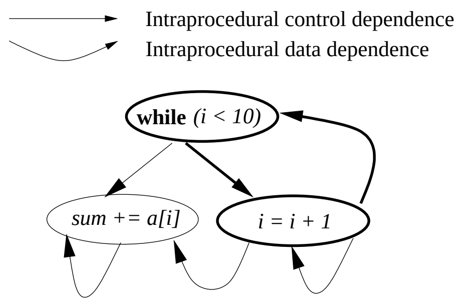
Intraprocedural control dependence
Intraprocedural data dependence
while (i < 10)
sum += a[i]
i = i + 1

Fig. 1. Intraprocedural SCC Example. The two bold vertices form an SCC.

Original Program

```
int main()
{
  sum = 0
  i = 1
  while i ≤ 10
    sum = add( sum , i)
    i = add(i, 1)
  print sum
  print i
}

int add(a, b)
{
  return a + b
}
```

Slice on "**print** *i*"

```
int main()
{

  i = 1
  while i ≤ 10

    i = add(i, 1)

  print i
}

int add(a, b)
{
  return a + b
}
```

Fig. 2. A slice computed with respect to (the vertex representing) the value of i at the statement "**print** i".

flow dependence edge runs from a vertex that represents a definition of a variable to a vertex that represents a use of the variable reached by that definition [Horwitz et al. 1990; Ferrante et al. 1987]. The dependence edges are safe approximations to the semantic dependences found in the program [Podgurski and Clarke 1990], which are generally not computable and therefore must be (safely) approximated. For example, Figure 1 shows a fragment of a procedure and the corresponding PDG fragment including both control and data dependence edges.

Taken with respect to vertex v, a slice includes the vertices that represent statements that potentially affect the computation represented at v. Figure 2 shows a simple program and the statements corresponding to the vertices in one of its slices. An interprocedural slice can be computed using simple graph reachability. However, the resulting slice is imprecise as it fails to take calling context into account. For example, both calls to *add* from Figure 2 would be included in such a slice. A two-pass reachability algorithm [Horwitz et al. 1990] avoids this imprecision.

In more detail, to address the calling-context problem, an SDG includes *summary* *edges*, which represent *transitive* dependences due to procedure calls. A summary edge at call-site c connects actual-in vertex v to actual-out vertex u if there is a path in the SDG from v to u that *respects calling context* by matching calls with returns (*i.e.*, there is an interprocedurally realizable path from v to u). For the slice taken with respect to vertex v, the two-pass algorithm first identifies the necessary vertices in v's procedure P and in procedures calling P. It does this without descending into called procedures (by ignoring interprocedural edges from formal-out to actual-out vertices). This avoids the difficulty of identifying to which call site analysis should "return." The second pass identifies the necessary vertices from called procedures by descending only into procedures, again side-stepping the calling-context problem.

3.2 The Subjects

The study considers just over 1 million lines of C code from 43 subject programs that range in size from 563 to 167K lines-of-code. Three programs were obtained from the European Space Agency (ESA), while the remainder are publicly available. Figure 3 presents summary information for these programs. It includes the size of each subject as measured in lines of code (LOC), the size of the resulting SDG (given by both vertex and edge counts), and finally the number of slices computed for each program.

Note that the number of lines-of-code and the number of slices do not match for two reasons. First, lines-of-code were counted using the Unix word count utility which is a rather crude method. Word count was chosen to make the results comparable with other studies that report lines-of-code as counted by word count [Anderson et al. 2001; Eisenbarth et al. 2002; Krinke 2002]. Second, a slice was computed for every vertex that represents executable source code. One "line of code," as counted by word count, may be represented by multiple vertices. For example, the single line "`if (a && b)`" is represented by two vertices to correctly model the sequence point and short-circuit (conditional) evaluation of boolean expressions in `C`.

The subject programs cover a wide range of programming styles. For example, the program `prepro` is Fortran-esque in its use of arrays. In contrast, several other programs make heavy use of function pointers. The program `ed` is rather tight knit and "single minded." In contrast, the program `acct` contains many different (related) computations.

3.3 Threats to Validity

With any empirical experiment, it is important to consider threats to validity. In the absence of human subjects, only two potential threats to validity need to be considered. These are threats to external validity and internal validity.

External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized. In this experiment, a single platform and SDG construction algorithm were used and it is possible that the selected programs are not representative of programs in general. Thus, the results might not transfer to other platforms or SDG constructions (*e.g.*, using an alternate points-to analysis) and may not apply to "typical" programs.

The use of other slicing algorithms might lead to different results. For example

name	LOC	vertices	edges	slices
a2ps	53,900	417,513	6,303,686	58,009
acct-6.3	9,536	18,531	80,584	7,222
barcode	5,562	10,916	35,796	3,824
bc	14,609	14,903	49,523	5,105
byacc	6,337	34,019	114,377	10,150
cadp	11,068	42,212	122,478	15,672
compress	1,234	5,211	14,749	1,085
copia	1,170	43,513	126,943	4,680
csurf-pkgs	36,593	295,220	973,703	42,777
ctags	16,946	120,014	422,034	20,313
cvs	93,309	6,219,909	28,981,757	102,353
diffutils	18,374	36,079	195,348	16,622
ed	12,493	44,387	222,858	16,368
empire	53,895	496,033	2,022,295	105,918
epwic-1	8,631	22,217	79,026	12,447
espresso	22,050	93,326	467,165	29,044
findutils	16,891	30,322	170,697	14,320
flex2-4-7	15,143	39,037	186,523	11,104
flex2-5-4	20,252	55,019	386,413	14,114
ftpd	15,914	56,981	366,095	24,820
gcc.cpp	4,079	19,095	89,513	7,354
gnubg-0.0	7,229	25,138	80,922	9,447
gnuchess	16,659	44,957	163,465	15,069
gnugo	15,217	278,766	1,044,525	58,373
go	28,547	111,246	416,404	35,594
ijpeg	24,822	62,698	265,650	23,954
indent	6,100	25,737	582,728	6,602
li	6,916	889,260	3,762,858	13,521
named	103,670	1,326,070	21,824,057	104,567
ntpd	45,647	204,420	804,652	39,573
oracolo2	14,326	20,654	65,734	10,846
prepro	14,328	20,578	65,353	10,779
replace	563	1,097	3,500	867
sendmail	75,156	1,111,054	37,774,280	46,889
space	9,126	20,018	64,537	10,311
spice	149,050	1,774,846	31,308,396	213,625
termutils	6,697	8,869	27,442	3,096
tile-forth	3,717	59,247	215,288	11,940
time-1.7	6,033	4,583	14,092	1,049
userv-0.95.0	7,150	70,796	272,726	12,511
wdiff.0.5	5,958	7,213	21,793	2,421
which	4,880	4,666	13,327	1,156
wpst	17,321	54,658	209,465	20,667
sum	1,007,098	14,241,028	140,412,757	1,176,158
average	23,421	331,187	3,265,413	27,353

Fig. 3. Characteristics of the subject programs studied.

using a less precise points-to algorithm would have a significant, though predictable effect. Considering 43 subject programs helps to mitigate the second concern. The diversity of these programs makes it more likely that the conclusions about the techniques generalize. However, as most of these are open-source programs, it remains possible that non-open source programs would exhibit significantly different behavior.

Internal validity is the degree to which conclusions can be drawn about the causal effect of the independent variable on the dependent variable. In this experiment, the only serious threat comes from potential for faults in the seven slicers studied. Other common forms of internal validity, for example construct validity (the degree to which the variables used in the study accurately measure the concepts they purport to measure) are not an issue as the variables measured (*e.g.*, slice size and time), can both be measured with high precision. Thus, the only serious internal validity concern comes from the assumption that slicers correctly implement each slicing algorithm. In practice, the slicing tools might contain errors, or employ imprecise analyses (*e.g.*, imprecise data-flow analysis or imprecise points-to analysis). To mitigate this concern, for each program the output of the seven slicers was compared (and found identical in all cases). Thus, any slicing fault is present in all of the slicers. This reduces the impact implementation faults may have on the conclusions reached.

3.4 Data Collection

The data was collected on a dual processor Linux box running Red Hat Linux 7.1/kernel version 2.4.2-2. Each processor is a 1000MHz Pentium III and has a 256Kb cache with a 32 byte cache line. The processors share 4Gb of main memory. Memory contention between the two processors slows each down by 15.5% relative to a single processor in an identical environment. CPU utilization was $99^{+}\%$ for all runs of each slicer. Each slicer was built using the GNU compiler, `gcc` version 2.96 with `-O3` optimization.

Data was collected by slicing on every vertex in a program's SDG that directly represents source code. The most common cause of vertices that do not directly represent source code are vertices that represent the passing of global variables to and from procedures. Including the slices on these non-source vertices does not affect the results, other than providing more data for each program. Finally, slices were not taken with respect to vertices representing the standard C libraries `libc` and `libm`.

4. TECHNIQUES

This section introduces the six optimization techniques investigated and describes how they are combined to form the seven slicers studied in the next section. The six are summarized in Figure 4. They include four algorithmic techniques: the formation of three kinds of strongly connected components and topological sorting, or top sorting, (labeled SCC, iSCC-s, and iSCC-c, and TS, respectively), and two "low level" techniques: vertex size reduction and redundant transitive edge removal (labeled Pack and Tran, respectively). These six are all preprocessing steps; thus, other than the time taken, they do not affect the two-pass slicing algorithm.

The seven slicers are based on Std, the standard (unoptimized) slicer. Std is a

Version	Description
SCC	intraprocedural SCC formation
iSCC-s	separate interprocedural SCC formation
iSCC-c	combined interprocedural SCC formation
Pack	memory packing and field layout
TS	topological sorting of vertices
Tran	transitive edge removed

Fig. 4. The six techniques studied.

straightforward implementation of the two-pass interprocedural slicing algorithm presented by Horwitz et al. [Horwitz et al. 1990]. The vertex data structure and excerpts from the slicer's implementation are shown in Figure 5. The function `b_slice` drives the computation. It marks the vertices of Pass 1 by calling function `b1_flood`, which avoids descending into called procedures by considering only interprocedural edges into formal-in and entry vertices. These edges connect the procedure to calling procedures. The actual-out vertices identified in Pass 1 form the starting point for the second pass. Function `b_slice` calls Function `b2_flood` (not shown) starting from each actual-out vertex encountered during Pass 1. Function `b2_flood` is similar to `b1_flood` except interprocedural edges into called procedures are traversed rather than those out to calling procedures.

The six optimization techniques exploit certain patterns in the dependence graph in an attempt to reduce slice computation time. The first three techniques form different kinds of *Strongly Connected Components* (SCCs). All three satisfy two key requirements: first all vertices in an SCC will have the same slice and, second, any slice that includes a vertex from an SCC will include all the vertices of the SCC. Satisfying these two requirements allows each SCC to be collapsed into a single representative node. This is an instance of a minimal, consistent, and sharp vertex coarsening [Harman et al. 2001], which means that the slices produced are *identical* to those that would have been produced without merging vertices into SCCs.

The first SCC formation technique, denoted by SCC, builds intraprocedural SCCs by ignoring interprocedural edges during SCC construction. Because slicing within a procedure amounts to computing a transitive closure, SCC clearly satisfies the two requirements. An example of an intraprocedural SCC can be seen in Figure 1 where the vertices labeled "while $(i < 10)$" and "$i = i + 1$" form a two-vertex SCC. Any slice that includes one of these vertices will also include the other: thus, the two vertices can be treated as one. Intraprocedural SCCs are the simplest and most efficient of the three SCC formation techniques to compute.

The remaining two SCC techniques consider interprocedural edges. Such SCCs are created as a result of recursion or when a procedure call occurs in a loop where results from one call flow into a call in a subsequent loop iteration. The first interprocedural SCC technique is *Separate Interprocedural SCC*, denoted by iSCC-s. In some sense this technique clones and then creates two specialized versions of the SDG. The clones are specialized to match the two passes of the interprocedural slicing algorithm. Thus, a vertex may be in two separate SCCs, one that is used during Pass 1 of a backward interprocedural slice and a second that is used during Pass 2 of a backward interprocedural slice. Of the three techniques, this produces

```
typedef
{
  int  name;
  int  mark;
  int  kind;
  PdgVertex *outgoing_intra_edges;
  PdgVertex *incoming_intra_edges;
  PdgVertex *incoming_inter_edges;
  PdgVertex *outgoing_inter_edges;
} *PdgVertex;

b_slice(PdgVertex from)
{
  // ignore non-source vertices
  if ((from->kind & FROM_LIBC) || (from->kind & NO_SOURCE))
    return;

  actual_outs = list_initialize();
  b1_flood(from);
  list_foreach(actual_outs, b2_flood);
}

b1_flood(PdgVertex v)
{
  if (v->mark != current_mark)
  {
    int kind = v->kind;
    PdgVertex *edges;
    v->mark = current_mark;
    slice_size++;

    edges = v->incoming_intra_edges;
    if (edges != NULL)
      while (*edges != NULL)
        b1_flood(*edges++);

    if (formal_in(kind) || entry(kind))
    {
      edges = v->incoming_inter_edges;
      if (edges != NULL)
        while (*edges != NULL)
          b1_flood(*edges++);
    }

    if (actual_out(kind))
      list_insert_beginning(actual_outs, v);
  }
}
```

Fig. 5. Std: The Unoptimized Version of the Slicer.

the largest SCCs.

For example, using the program shown in Figure 6, Figures 7 and 8 illustrate the pass specific SCCs formed by iSCC-s. Figure 7 illustrates the two SCCs found

```
int g()
{
  a = f(0);
  return a;
}

int f(int x)
{
  x = f(x);
  while (...)
  {
    b = x;
    x = f(b);
  }
  c = f(b);
  return x;
}
```

Fig. 6. Program used to illustrate SCC computation.

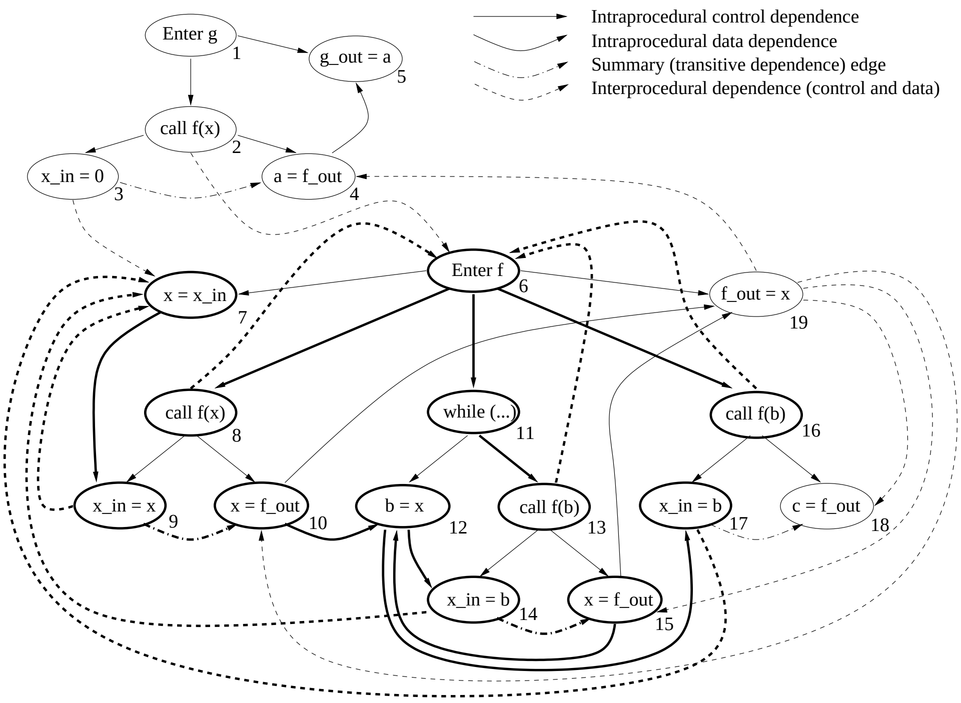

Fig. 7. Combined Interprocedural SCC Formation Example I

by iSCC-s for use during Pass 1: The bold vertices form two SCCs that include interprocedural edges only considered during Pass 1. Figure 8 illustrates an SCC found by iSCC-s for use during Pass 2: The bold vertices form an SCC that includes interprocedural edges only considered during Pass 2. Consider, for example the two

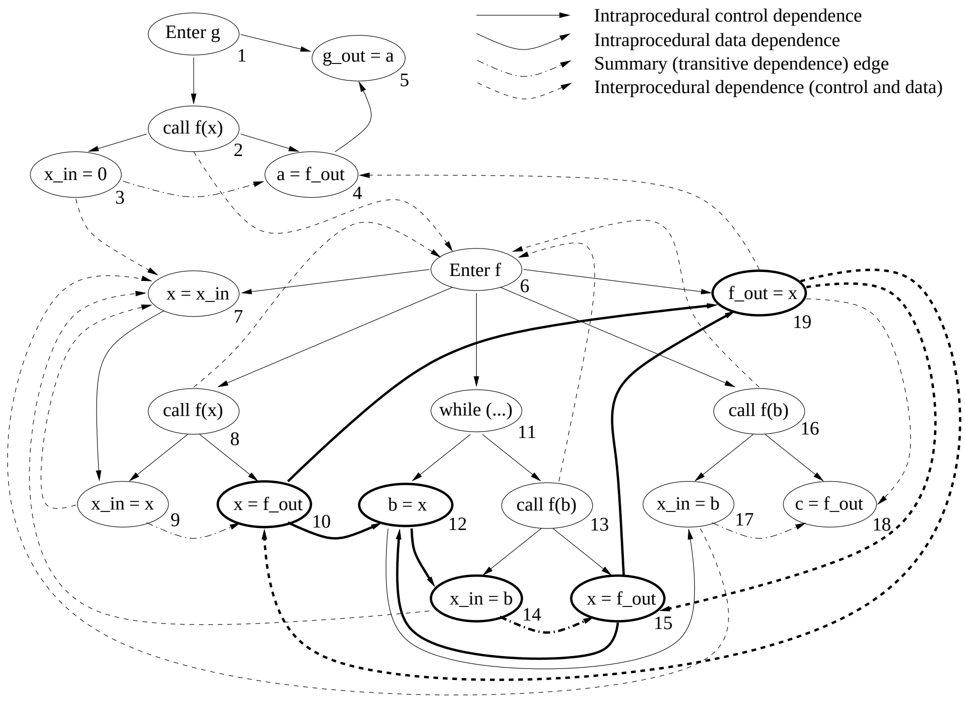

Fig. 8. Combined Interprocedural SCC Formation Example II

SCCs in Figure 7. One of them contains the Vertices 6, 8, 11, 13, and 16. It is due to the call edges (connecting Vertices 8, 13, and 16 to Vertex 6). The other SCC, which contains the Vertices 7, 9, 10, 12, 14, 15, and 17 is due to the parameter-in edges (connecting Vertices 9, 14, 17 to Vertex 7). If any bold vertex is encountered during Pass 1, all of the bold vertices of the corresponding SCC will be included in the slice. However, this is not true when one of these vertices is encountered during Pass 2. For example, consider the computation of a backward slice starting at Vertex 5. The slice will include only Vertices 1–5 during Pass 1 (the edge between Vertices 4 and 19 is not traversed during Pass 1). During Pass 2, this edge is traversed and some of the vertices in the SCC in Figure 8 are encountered (*e.g.*, Vertex 15). However, Vertex 16, for example, is not included in the Pass 2 slice. If the SCC in Figure 7 were also used in Pass 2, the slice would incorrectly include both complete SCCs, and Vertices 16 and 17 in particular, which should not be included (*e.g.*, the computation represented by Vertex 17 never influences the computation represented by Vertex 5).

The final SCC formation technique, denoted by iSCC-c, attempts to strike a balance between the first two. Unlike iSCC-s, its SCCs can be used by both slicing passes. However, unlike SCC, interprocedural edges are considered. This technique simulates the two passes of the slicing algorithm by intersecting the results of two SCC-building passes. Thus, its interprocedural SCCs must involve recursive calls. The first pass traverses only intraprocedural edges and edges reaching called procedures (ignoring edges from formal-out to actual-out vertices). The second pass

```
typedef
{
  int name;
  int mark;
  int kind;
  PdgVertex *incoming_intra_edges;
  PdgVertex *incoming_inter_edges;
  PdgVertex *outgoing_intra_edges;
  PdgVertex *outgoing_inter_edges;
  // fields to support SCC discovery
  int scc_size;
  int number;
  int low_link;
  PdgVertex scc_rep;
} *PdgVertex;
```

Fig. 9. SCC: Vertex Declaration

builds SCCs along intraprocedural edges and edges leaving called procedures (edges from formal-out to actual-out vertices). The final SCCs are built from the nodes that belong to the same SCC in the first and the second passes. The final SCCs are not true SCCs, but they fulfill the two requirements.

For example, Figures 7 and 8 illustrate the SCCs found during Pass 1 and Pass 2. Vertices 10, 12, 14, and 15 are in the same SCC during Pass 1 and Pass 2. Thus, they form an untrue SCC that can be used both during passes 1 and 2. Whenever one of these four vertices is included in a Pass 1 or Pass 2 slice, the other three will also be included. In comparison, with intraprocedural SCC formation only Vertex 10 has been additionally included in an SCC.

The implementation of the three SCC techniques is similar. The first, SCC, includes a straightforward implementation of intraprocedural SCC construction [Fischer and LeBlanc 1988]. As seen in Figure 9 intraprocedural SCC formation increases the size of a vertex by four additional fields, which has a negative impact on slicer performance. The creation of separate interprocedural SCCs by iSCC-s further increases the vertex size as it requires replacing the field `PdgVertex scc_rep` with two fields: `tPdgVertex pass1_scc_rep` and `tPdgVertex pass2_scc_rep`. Finally, forming combined interprocedural SCCs by iSCC-c adds a field `scc1`, which is used to map the nodes to SCCs of the technique's first pass for comparisons in the second pass.

In addition to the three SCC techniques, the final algorithmic technique changes the order of the vertices in memory. This is achieved by splitting the implementation into two phases. The first phase writes the vertices out in the desired order. The second phase reads them into an array, thereby preserving vertex order. Three orders were considered: depth-first-search order, breath-first-search order, and topological-sort order. The first two produced no significant improvement from the random order in which vertices are initially read in. However, topological sorting proved beneficial; thus, the fourth algorithmic technique, denoted by TS, topologically sorts the vertices of each PDG. Note that, intraprocedural SCC formation is a necessary preprocessing step to topological sorting, since a precondition

```
typedef
{
  PdgVertex *incoming_edges;
  int scc_size;  // negative -> scc_rep;
  int kind;
  int mark;      // holds low_link
                 // during scc computation
  PdgVertex *outgoing_edges;
  int name;      // holds 'number' during
                 // scc computation
} *PdgVertex;
```

Fig. 10. Pack: Vertex Declaration

to the sort is that its input be a DAG.

The final two techniques are "low level" techniques. The first, Pack, employs bit packing to reduce the memory footprint of a vertex by packing structure fields and exploiting cache behavior. In particular, the four edge pointers of Std (see Figure 9) are collapsed to two edge pointers (as shown in Figure 10). This is possible because the two low-order bits of a pointer on the architecture used in the study are always zero; thus, the edge kind (intraprocedural or interprocedural) can be "packed" into these bits. After packing, the core of the function `b1_flood` becomes

```
for( ; *edges != NULL; edges++)
  if (marked_traverse_in_pass1(*edges))
    b1_flood(strip_edge_mark(*edges));
```

This is an example of a classic time-space trade off. It produces extra work at run-time in the form of packing and unpacking in exchange for reducing vertex size. However, the relative speed of the memory and the processor make the packing and unpacking essentially "free." This relative speed difference exists on all modern processors. Thus, assuming bits could be found, other processors would be expected to show similar improvement.

Pack also pays attention to two memory layout issues. First, related fields of a vertex structure are constructed to be adjacent in an attempt to place them in the same cache line. For example, on the target architecture kind and mark are now always in the same cache line. This is not the case using the declaration shown in Figure 9. Second, vertices' incoming and outgoing edges are allocated from separate heaps, thereby improving locality of reference in main memory.

The final technique, Tran, performs transitive edge removal. For example, a code sequence such as

```
a = 1;
b = a;
c = a + b;
```

includes a "redundant" flow dependence edge: There are flow dependence edges from `a = 1` to `b = a` and `c = a + b` and from `b = a` to `c = a + b`. In terms of reachability, the edge from `a = 1` to `c = a + b` is redundant. The importance of this observation is easier to see from the perspective of the definition of dependence graph slicing than from the perspective of the computation of a slice, which typically involves a backward flood along the edges of the dependence graph. The definition of the slice on Vertex v is the set of vertices from which v is reachable [Horwitz et al. 1989; Horwitz et al. 1988; Reps et al. 1994].

Transitive edge removal is a compromise between two extremes. At one extreme, edges could be added to transitively close the graph, thereby reducing reachability to adjacency (*i.e.*, connected by an edge). Adding these edges would speed reachability (at a tremendous space cost). The other extreme is to perform minimal transitive reduction, which yields the least graph whose transitive closure is the same as the transitive closure of the input graph. Finding a minimal transitive reduction is NP–complete and thus, is prohibitively expensive to compute. The resulting graph would require less memory but cause no increase in slice computation time.

Tran represents an easy-to-compute space-efficient compromise between these two extremes. For each vertex v in the graph, the following two-step procedure is performed. First, all vertices to which v has an outgoing edge are marked as *directly reachable*. Then, staying within the PDG (*i.e.*, considering only intraprocedural edges), edges are walked starting from each of these vertices in a forward direction. If a vertex w marked *directly reachable* is encountered during this walk, then the edge from v to w is removed as a transitive edge. This process takes $O(N * n)$ time, where N is the number of vertices in the SDG and n is the number of vertices in the largest PDG. Removing these edges saves the space needed to store them and the time needed to traverse them during slicing.

The six optimization techniques are combined to form seven different slicers. This is a small fraction of the possible number of slicers that could be produced. For example, if all six techniques were independent, 6! possible slicers could be formed. However, the techniques are interrelated. For example, as noted earlier, topological sorting requires the removal of (intraprocedural) cycles and thus must follow one of the three SCC techniques. Also, only one of the three SCC techniques can be applied, as all are mutually exclusive. Even after taking these interrelations into account, there are still too many combinations to run all possible slicers. The seven slicers summarized in Figure 11 were chosen for study. They cover the primary interesting combinations and are described in the remainder of this section.

The first slicer, Std, is included to compute base-line data. The next four incorporate the techniques SCC, Pack, TS, and Tran to form a chain of increasingly optimized slicers. In more detail, the SCC optimization technique is used to augment Std producing +SCC. Next, the "low-level" packing algorithm is used to augment +SCC resulting in +Pack. This version is further augmented by topological sorting to produce +TS. Finally, transitive edge removal is applied, to produce the fully optimized version, +Tran. The final two slicers are produced by replacing SCC, the intraprocedural SCC computation, with one of the two interprocedural SCC formation techniques. The first, w/iSCC-c, replaces SCC with iSCC-c, while the second, w/iSCC-s, is a separate pass applied after intraprocedural SCCs are formed by SCC.

Version	Description
Std	Unoptimized "standard" implementation
+SCC	Std with SCC formation
+Pack	+SCC incorporating memory packing and field layout work
+TS	+Pack with vertices sorted topologically in each procedure
+Tran	+TS with transitive edges removed
w/iSCC-c	+Tran with SCC replaced by iSCC-c
w/iSCC-s	+Tran with SCC replaced by iSCC-s

Fig. 11. The Seven Versions of the Slicer.

5. RESULTS AND ANALYSIS

This section describes the results of running each of the seven slicers described in Figure 11 on the programs described in Section 3. Summary data covering all executions is first discussed, followed by an examination of data specific to each different slicer (such as the number and size of the different SCCs). A discussion of memory use appears at the end of the section.

Overall, the combination of optimization techniques produces an average four-fold performance increase. Each individual technique produces an average increase in performance in its own right. However, for the two approaches to interprocedural strongly connected components, although there are performance increases, there are some programs which do not benefit from some of the strongly connected component techniques. The results indicate that the behavior of the graph reachability algorithm, optimized according to strongly connected components, is highly program-specific. This may suggest an alternative approach to cohesion measurement, in terms of the nature of the dependence graph's strongly connected components. However, this remains a topic for future study, as it is outside the scope of the present paper.

In this section, relative slice computation times for all seven slicers are shown in Figure 12. This same data is shown graphically in Figure 13. The times are scaled so that the running time for the unoptimized version (Std) is 1.00 for each program. This was necessary because the raw slicing times for Std ranged from 0.05 seconds to 768,391 seconds, making summary statistics, such as averages, meaningless (for reference, the unscaled times for +Tran are shown in Figure 14). In Figure 12, the last three rows show the sum, average, and the total percent run-time reduction. As can be seen in the data, the average performance increases as each of the first four techniques is incorporated into the slicer. Combined, the first four techniques produce an average 71% time reduction. The final two columns in Figure 12 show the performance increase when SCC is replaced by iSCC-c and iSCC-s, respectively. Both produce a mild improvement over +Tran. Figure 12 includes the performance increases for backward slicing. In general, the performance increases for forward slicing are slightly better (about 4% faster for each slicer), but overall show the same trend.

The performance of each slicer is now considered. First, the implementation of +SCC allocates the fields related to SCC formation as part of a vertex, rather than as a separate table. This has the advantage of being faster since a separate table

Program	Version						
	Std	+SCC	+Pack	+TS	+Tran	w/iSCC-c	w/iSCC-s
a2ps	1.00	3.33	4.59	6.64	7.13	7.75	9.15
acct-6.3	1.00	1.54	2.74	4.35	5.70	5.82	6.91
barcode	1.00	0.99	2.15	3.62	5.51	5.47	6.59
bc	1.00	1.06	1.84	3.10	3.88	4.85	4.54
byacc	1.00	1.38	2.08	3.28	3.58	3.95	4.48
cadp	1.00	1.14	1.63	2.94	3.28	3.46	3.96
compress	1.00	1.25	1.67	1.67	1.67	2.00	2.50
copia	1.00	0.88	1.18	1.43	1.50	1.38	330.50
csurf-pkgs	1.00	0.82	1.11	1.39	1.47	1.86	2.41
ctags	1.00	1.17	1.60	2.36	2.57	2.77	3.27
cvs	1.00	0.86	1.07	1.56	1.73	3.43	3.87
diffutils	1.00	1.80	2.68	3.93	4.43	4.95	5.81
ed	1.00	1.83	2.62	3.72	4.20	4.85	5.52
empire	1.00	1.32	1.82	2.57	2.79	3.14	3.60
epwic-1	1.00	1.14	2.32	5.32	7.36	7.51	9.11
espresso	1.00	1.18	1.59	2.25	2.48	2.84	2.92
findutils	1.00	1.56	2.48	3.94	4.56	5.29	5.43
flex2-4-7	1.00	1.63	2.29	3.16	3.37	3.78	4.23
flex2-5-4	1.00	2.34	3.23	4.66	5.29	5.67	6.23
ftpd	1.00	1.79	2.44	3.24	3.44	3.97	4.21
gcc.cpp	1.00	1.48	2.37	3.89	4.62	5.99	6.04
gnubg-0.0	1.00	1.00	1.52	2.31	2.64	2.88	2.98
gnuchess	1.00	1.34	1.91	2.79	3.25	3.52	3.69
gnugo	1.00	0.93	1.32	1.73	1.84	2.25	2.41
go	1.00	1.13	1.87	2.37	2.63	2.92	3.08
ijpeg	1.00	1.18	1.79	2.22	2.57	3.68	3.45
indent	1.00	5.40	7.85	11.59	14.01	14.96	15.44
li	1.00	0.90	1.20	1.62	1.77	3.29	4.93
named	1.00	1.35	1.66	2.16	2.25	3.15	5.08
ntpd	1.00	1.10	1.49	1.90	2.13	2.29	3.05
oracolo2	1.00	0.87	1.43	2.19	2.51	3.01	3.26
prepro	1.00	0.89	1.43	2.44	2.65	2.80	3.47
replace	1.00	1.67	2.50	2.50	2.50	5.00	5.00
sendmail	1.00	1.67	1.82	2.33	2.36	3.53	6.15
space	1.00	0.89	1.50	2.38	2.69	2.98	3.46
spice	1.00	1.02	1.52	1.80	1.80	1.99	2.72
termutils	1.00	1.09	3.15	4.31	5.28	5.41	6.39
tile-forth	1.00	1.00	1.41	1.80	1.97	2.43	2.84
time-1.7	1.00	1.25	2.50	1.67	1.67	1.67	1.67
userv-0.95.0	1.00	1.02	1.44	1.91	2.06	2.41	3.04
wdiff.0.5	1.00	1.56	1.87	1.87	2.33	2.55	2.80
which	1.00	1.33	1.50	1.71	2.00	2.00	3.00
wpst	1.00	1.19	1.72	2.45	2.75	3.12	3.36
sum	43.00	60.28	89.91	127.07	146.24	168.58	522.56
average	1.00	1.40	2.09	2.96	3.40	3.92	12.15
percent reduction		29%	52%	66%	71%	74%	92%

Fig. 12. Backward Slicing Speedup (scaled to Std = 1.00). (Excluding the outlier copia from w/iSCC-s, the total, average and percent reduction for the final column are 192.06, 4.57, 78%, respectively.)

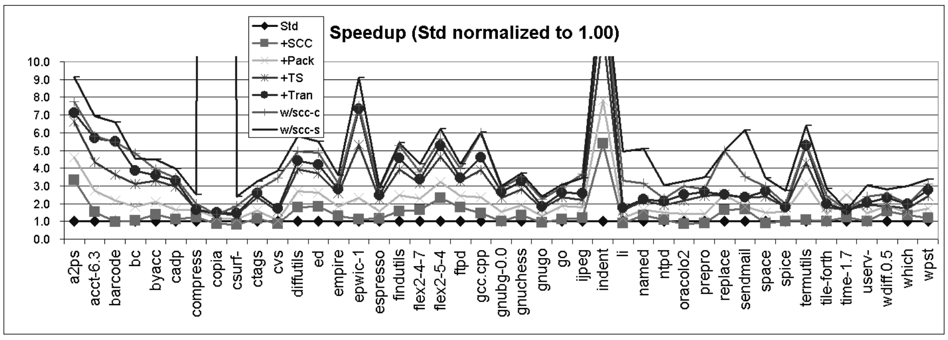

Fig. 13. Performance Increase of Backward Slicing for All Seven Slicers on All Programs.

lookup is not required. It has the disadvantage that fields for the SCC computation cannot be easily deallocated once SCCs have been formed. The impact of this can be seen in the data from the Column +SCC of Figure 12: SCC formation is not beneficial for all programs. The worst case, `csurf-pkgs`, has a performance increase of 0.82 (*i.e.*, runs 18% slower). Lack of a universal benefit is caused by two conflicting effects on run-time. First, the increase in vertex size required to maintain data used by SCC formation tends to increase computation time as it negatively affects cache performance. On the other hand, not repeatedly processing the vertices that make up each SCC tends to decrease computation time. Over all programs studied, SCC formation causes a net performance increase, as seen in the last three rows of the table.

Next, consider the +Pack column, which shows the effect of the two related packing techniques: the vertex size reduction and structure field reordering. On the target architecture, pointers and integers take four bytes: thus, a vertex from the Std version takes 28 bytes and a vertex from the SCC version takes 44 bytes (iSCC-s and iSCC-c require 48 and 52, respectively). In comparison, a vertex from the Pack version takes only 24 (respectively 28 and 32 for iSCC-s and iSCC-c). This reduction was achieved by reducing the number of edge pointers and by reusing certain fields for different purposes during different phases of the computation. For example, the field `mark` used to mark vertices as "in a slice" during the slicing phase, is also used to hold `low_link` (see Figure 9) during SCC formation. The total memory used by some of the larger subject programs is discussed at the end of the next section.

In comparison with +SCC, size reduction accounts for three quarters of the improvement seen with +Pack and layout is responsible for the remaining quarter. For +Pack, the smallest performance increase is 1.11 and the largest is 7.85. The rather large spread between these figures suggests future work on why SCC formation is more effective in some cases than others. In particular, this might say something interesting about the underlying cohesiveness of programs and give general guidelines for program development.

Packing makes it possible to isolate the effect of SCC formation by reusing the fields of a vertex so that SCC formation does not need any additional fields. In other words, it forms SCCs without changing the size of a vertex (as +SCC does). This can be done by considering only the vertex size reduction (and ignoring the reordering), thereby obtaining the true effect of SCC formation, without the confounding effect of node size. Doing so, the average performance increase is 1.92 (compared to 2.09 when including the reordering). More importantly, each of the 43 programs shows some reduction. Thus, given constant vertex size, SCC formation always leads to an improvement.

As seen in Figure 10, `Pack` biases the computation in favour of operations that use `incoming_edges` over those that use `outgoing_ edges` because the latter is in a different cache line from the `scc_size`, `kind`, and `mark` fields (the `name` field holds a unique vertex-id, and is not used after the SCC formation). This bias shows up in the data. The overall improvement for backward slicing, which traverses incoming edges is 14%, while the overall improvement for forward slicing is only 7% (the first 7% is accounted for by the bit packing in the edge lists).

Next, consider the column for top sorting (the +TS slicer). To facilitate placing vertices in topological order, the slicer +TS includes two phases (run in separate processes). The first forms SCCs, topologically sorts, and sends the vertices to the second phase, which computes the slices. During slicing, this allows further reduction in vertex size from 20 to 16 bytes as memory for fields required only during SCC formation need not be allocated during Phase 2.

Slicer +TS provides another significant improvement. This improvement has two main constituents: the top sorting and a drop in vertex size. While not shown in the figure, top sorting accounts for 86% of the reduction. This result suggests future work on techniques for automatically laying out data structures. Such work would mirror similar "lower-level" layout work [Chilimbi et al. 1999].

The only slowdown with this version is for program `time-1.7` whose performance increase goes from 2.50 to 1.67. Program `time-1.7` is the second smallest program. The times taken to slice these small programs approach the resolution of the system timer. Thus, this apparent reduction is in fact an artifact of timer's resolution.

Column six shows the performance increases for +Tran, which incorporates transitive edge removal. Like +TS, the slicer +Tran includes two phases. The first forms SCCs, topologically sorts, marks transitive edges for deletion, and sends the vertices to the second phase, which computes the slices. Edges marked for deletion are never written out by Phase 1 and thus are never considered by Phase 2. At first glance, transitive edge removal appears to be another time-space trade off. In other words, the removal of transitive edges should save the space used to store transitive edges at the expense of having to rediscover the path's transitive edges. However, for slicing, the removal of these edges saves both time and space. The explanation for this behavior lies in the realization that the transitive edges in the SDG have no exploitable pattern; therefore, the slicer cannot exploit their presence.

Over all 43 programs, an average of 25% of the intraprocedural edges were removed as redundant transitive edges, producing a 15% run-time improvement. Although this is considerably less than the 40% increases for each of SCC formation and topological sorting, it is still worthwhile.

Having large procedures and in particular one large procedure also affects the other techniques. The effect of TS is amplified by having large functions because topological sorting is done on a per-procedure basis: thus, programs with lots of small functions get less benefit than those with fewer larger functions. This is clearly the case with indent's large function. A similar observation applies to Tran which does not consider interprocedural edges and thus would be expected to work (slightly) better in the presence of large functions.

For indent, vertex packing provides the average benefit, but the program is small and its SDG may fit entirely in the L2 or even the L1 cache. Over all programs, the improvement caused by Pack also has the lowest standard deviation of all the techniques. Inspection of several programs reveal no source pattern that explains the variation in Pack. A particularly poor cache layout that improved because vertices shrank and thus changed cache lines is the leading contender.

The worst overall improvement was for csurf-pkgs followed closely by copia. Both of these include primarily small functions. Small functions do not support the formation of large intraprocedural SCCs. In addition, they reduce the effectiveness of TS, as sorting the vertices of a small function provides little advantage within the PDG, and the order of the PDGs is essentially random. The same is true for Tran to a lesser extent. As described below, the functions of copia are largely mutually recursive (csurf-pkgs' are not), which does create large interprocedural SCCs.

Finally, the two alternate SCC algorithms provide additional improvement when used to replace SCC in +Tran. Over all programs, w/iSCC-c provides a 15% improvement over +Tran. Excluding the performance increase for copia, w/iSCC-s provides a further 16.5% improvement over w/iSCC-c (the 50% improvement for which in Figure 12 is also a "timer-resolution" artifact). Copia is an outlier. Its performance increase of 330 is due to a 13,040 vertex Pass 1 SCC that is included in 4,461 of the 4,680 slices. As the size of these slices ranges between 13,000 and 14,000 vertices, their computation is very fast (the large SCC is likely to reside in the L1 cache during the entire computation).

The programs that received the best and the worst overall speedup using +Tran are now considered. In particular, program characteristics that cause the extremes are discussed. The program indent shows the best overall improvement (a speedup of 14.01). An inspection of its source code reveals that the main function includes a large (over 1400-line or almost 25% of the code) loop that processes input tokens. This loop includes a "switch on type code" case statement with over 1150 lines. Because this code is all in one function it is possible for it to be part of a large intraprocedural SCC. This indeed happens as the processing of various type codes influence each other, resulting in an SCC that includes about one third of the SDG: 47.5% of indent's 25,737 vertices that belong to SCCs including one of 8,623 vertices (from the main loop).

An inspection of the source code for those programs that had large SCCs reveled two causes. First, programs such as indent, with a few disproportionately large functions, support the formation of large intraprocedural SCCs. The second cause produced the largest improvement of any of the techniques. The reduction is for w/SCC-s when applied to copia. While copia has few global variables and small functions, it essentially implements a finite state machine with a central dispatch function that calls most other functions and is called by all of these. This recursion leads to large interprocedural SCCs and thus the tremendous speedup. The vertices of these SCCs are mostly procedure inputs and outputs that are treated separately by iSCC-s; however this means that iSCC-c does not discover large SCCs nor does

SCC.

The observation that some programs have very large SCCs is important not only for optimization purposes. It has practical significance, since large clusters of dependence can be an inhibitor to continued maintenance. This observation motivated a study of dependence clustering using massive slicing, which revealed a striking prevalence of large dependence clusters in the code base, suggesting opportunities for refactoring to reduce this 'dependence pollution' [Binkley and Harman 2005b].

The remainder of this section considers data related to each slicer independently. To begin with, for the +Tran slicer, two additional views of the data are presented. First, the following table shows the time taken (averaged over all programs) by each step in the computation. Note that slicing includes the time to compute both backward and forward slices [Horwitz et al. 1990] for every statement in the program. Average time per slice over all programs is 0.32 seconds while the average for `cvs`, which has the highest average, is 5.5 seconds.

Step	Percent Time Taken
Read in Graph	1.7%
SCC Computation	1.3%
Top Sorting	0.1%
Transitive Edge Removal	0.9%
Slicing	96.0%

For +Tran, Figure 14 shows some additional statistics related to raw slicing times. These include the average slice size, the total time taken to compute the two-pass closure for every executable code vertex from the program, and the pace of this computation. This data provides targets for future tools performing related analyses. Note that the data does not include parsing and preprocessing time-just the slice computation time. The final three columns include the number of slices that must be taken to pay for the preprocessing time.

Figure 14 also shows the point at which the pre-processing time invested in applying the optimization techniques pays off. That is, it shows the number of slices which must be computed before both the unoptimized and fully optimized forms of slicing would take identical time. For most of the programs this is well below 10% of the total slices in the program, making the technique applicable for massive slicing in all these cases. For a few of the smaller programs–`compress`, `replace`, `time`, `wdiff` and `which`–the break even point is not reached for some time. However, these programs each have no more than 2,500 slices, which is fewer than 10% of the average number of slices for all programs studied. They are all small programs with few slices, relatively speaking. Clearly, for such programs with only a few slices, optimizations for massive slicing may not be worthwhile.

Columns 5 and 6 show the number of slices Std could compute in the time it takes +Tran to perform its preprocessing. Column 5 assumes that each slice is computed "from scratch", while Column 6 assumes the graph is read in only once. The final column shows the number of slices that would need to be computed in order for the faster slicer +Tran to take less time than Std, assuming both slices read the

Program	Average Slice Size (vertices)	Total Time (sec)	Pace (KLOC/sec)	Std Slices Computable during pre-processing: Including read	Std Slices Computable during pre-processing: Excluding read	Break Even count	Break Even as %
a2ps	237,041	3835.6	463	7.0	910	1,042	1.8%
acct-6.3	1,770	1.0	6,783	0.6	188	219	3.0%
barcode	4,010	1.5	5,315	0.6	52	62	1.6%
bc	8,986	7.3	6,169	0.6	22	30	0.6%
byacc	9,731	16.8	1,098	0.6	61	84	0.8%
cadp	4,738	13.4	1,451	0.4	85	119	0.8%
compress	655	0.1	2,805	0.4	413	964	88.9%
copia	13,034	22.1	74	1.4	90	202	4.3%
ctags	72,960	360.3	581	0.5	22	34	0.2%
cvs	4,872,236	280298.2	27	2.4	1,507	3,243	3.2%
diffutils	9,477	28.6	2,802	0.9	87	116	0.7%
ed	30,883	85.6	1,662	1.8	76	99	0.6%
empire	336,665	7664.5	505	1.7	177	269	0.3%
espresso	40,583	351.3	793	2.5	173	284	1.0%
findutils	10,160	25.2	3,215	0.4	32	41	0.3%
flex2-4-7	12,510	30.7	1,756	0.7	61	85	0.8%
flex2-5-4	9,971	46.8	1,108	0.8	66	83	0.6%
ftpd	27,788	157.2	1,225	0.8	46	61	0.2%
gcc.cpp	10,097	12.5	1,272	0.4	22	28	0.4%
gnubg-0.0	9,520	20.9	1,239	0.4	25	39	0.4%
gnuchess	22,385	73.5	1,701	0.7	31	42	0.3%
gnugo	145,428	3145.9	147	1.6	100	195	0.3%
go	92,147	785.7	1,071	0.9	28	43	0.1%
ijpeg	28,414	193.8	1,390	0.5	22	33	0.1%
li	684,492	5181.0	14	3.1	285	610	4.5%
named	987,310	100196.2	81	4.3	931	1,648	1.6%
ntpd	98,652	1452.7	600	1.8	106	188	0.5%
oracolo2	6,363	14.4	3,329	0.7	68	107	1.0%
prepro	6,243	13.5	3,471	0.8	67	103	1.0%
replace	423	0.0	9,417	0.5	173	289	33.3%
sendmail	820,749	53917.1	48	9.2	2,422	4,212	9.0%
space	6,494	13.6	2,240	0.7	62	94	0.9%
spice	849,030	425938.2	36	2.4	523	1,252	0.6%
termutils	2,126	0.4	12,424	0.5	89	110	3.6%
time-1.7	324	0.0	14,922	0.5	734	2,448	233.3%
userv-0.95.0	30,217	141.4	270	1.2	71	127	1.0%
wdiff.0.5	775	0.1	12,916	0.5	415	632	26.1%
which	811	0.1	16,337	0.4	311	578	50.0%
wpst	10,151	50.5	1,316	0.6	95	144	0.7%
average	223,936	20586.9	3,006	1.4	254	473	1.7%

Fig. 14. Time and pace for the +Tran slicer. The *Break Even* column give the number of slices that must be computed to pay for the preprocessing costs.

SDG in once during initialization. Ignoring the 5 smallest programs, for which the resolution of the timer makes the computation errant, the average is about 1% of the total slices taken. Thus, for a user computing individual slices (from scratch), the preprocessing is paid for on average before the 3^{rd} slice. For a user computing a collection of slices, the preprocessing is paid for after about 1% of all slices have been computed.

Total time, Column 3, was used to generate a linear model predicting the computation time based on a graph's size as measured by the number of vertices and the number of edges encountered during slicing. The resulting linear model is

$$Time\ (in\ \mu sec) = 0.102 \times V + 0.103 \times E$$

where V is the number of vertices encountered (a vertex is counted each time it is included in a slice) and E is the number of edges encountered (an edge is counted each time it is traversed). This model has an R value of 95.9, which implies an excellent linear fit. There are approximately 5.9 edges for every vertex. Using this factor, the two coefficients can be combined, yielding $Time = 0.710 \times V$. Inverting the coefficients yields the pace of the computation, which is approximately 1.4 million vertices per second. As there are approximately 14.1 vertices per line of code, this denotes a pace of approximately 100 KLOC per second.

Finally, Column 4 shows the computation pace for each program. Recall that this includes the slicing time only and not the time for parsing the input nor for building the dependence graph. The final row of this column gives an average pace of 3,006 KLOC per second. One reason why this figure is higher than the 100 KLOC is because it is an unweighted average, and the pace of several of the small subject programs is rather fast. The weighted average is 1,177 KLOC per second. Both figures should be treated as approximations.

Three SCC algorithms were studied. Figures 15, 16, 17, and 18 present statistics that compare the different kinds of SCCs formed. Figure 15 shows numbers for +SCC and w/iSCC-c. This includes the number of vertices in SCCs, the number of SCCs, and the number and percentage of the vertices per SCC. For +SCC, this latter value ranges from 0.2% to 47.5% with three programs having over 30% of their vertices in SCCs (`indent`, `ed`, and `empire`). It is interesting to note that only `indent` is in the top three when ranked by level of performance increase. Programs `a2ps` and `diffutils` with performance increases of 3.33 and 2.18, respectively, show greater improvement than `ed` and `empire`. This is explained by observing that, while they have fewer vertices in SCCs, these SCCs are contained in a larger number of slices and thus their smaller savings have a large frequency of occurrence.

In general, w/iSCC-c produces larger but fewer SCCs than +SCC: the average size goes from 29.7 to 142.1 and the average count goes from 1,709 to 769 SCCs. This causes the percentage of vertices in SCCs to rise from 16.9% to 19.2%. Not all programs experience a benefit in terms of performance increase: a program-by-program comparison shows that 18 of the 43 programs have the same SCCs as +SCC.

Figure 16 shows the statistics for the two passes of the third SCC algorithm, w/iSCC-s. Computing separate SCCs for Pass 1 and Pass 2 should create larger SCCs. The data supports this observation. In general, when compared to the SCCs computed by +SCC and w/iSCC-c, those computed by w/iSCC-s are larger and fewer in number. The difference can be dramatic. For example, copia's Pass 1 SCCs are over 235 times larger than with +SCC.

Excluding the outlier copia, when compared to +SCC the average SCC is 70%

larger for Pass 1 and 56% larger for Pass 2. When compared to w/iSCC-c, the average SCC is 45% larger for Pass 1 and 34% larger for Pass 2. At the other end of the spectrum, 8 of the smaller programs (*e.g.*, `compress` and `space`) have only intraprocedural SCCs and thus have the same size SCCs for all three techniques. Program `li` has the largest percentage of vertices in SCCs with 72.4% of the vertices in Pass 1 SCCs and 65.0% in Pass 2 SCCs. Both are significantly greater than the averages of 27.1% and 24.7% respectively.

The final two rows of the table in Figure 16 show the sum and average of each column. On average there are more vertices in Pass 1 SCCs than in Pass 2 SCCs (7,509,911 versus 6,851,449) and more Pass 1 SCCs than Pass 2 SCCs (49,337 versus 46,198). The number of vertices per SCC is slightly greater with Pass 1 (10,885.0 versus 10,282.5 for Pass 2). This is because there are more procedure inputs than procedure outputs. Thus, there are more interprocedural edges considered during Pass 1, which has the effect of including slightly more vertices.

SCC computation provides two further illustrations of how excessive preprocessing can become prohibitively expensive. First, consider the costs and benefits of the three SCC formation techniques. Figure 17 presents the relevant data. The middle section of the table shows the computation time for SCC formation in the slicers +SCC, w/iSCC-c, and w/iSCC-s. The last two columns show the percentage increase in SCC formation time when SCC is replaced by iSCC-c and iSCC-s. Only programs for which one of the three SCC formation techniques took more than five seconds are shown. For the rest, the comparison is less interesting, as they are all "fast". For reference, the data includes the backward slicing time of the +Tran slicer, which incorporates the intraprocedural SCC technique SCC. As can be seen in the table, SCC takes negligible time relative to the time taken to compute the slices. This is not true of the two interprocedural SCC formation techniques. In fact, for some of the larger programs, forming the SCCs takes longer than slicing, making it inappropriate as an optimization technique for slicing.

The second illustration considers the formation time for intraprocedural SCCs by +SCC and w/iSCC-s, which computes intraprocedural SCCs as a preprocessing step. The data in Figure 18 show how w/iSCC-s takes longer to compute the same intraprocedural SCCs. This increase is caused by a vertex size increase. Comparing the columns labeled "SCC" and "iSCC-s intra only" allows the impact of an increase in vertex structure size to be assessed on the SCC computation (rather than on slicing as seen in Figure 12). The additional cost for iSCC-s varies from 7% to 1,826% and illustrates the impact cache effects have on graph algorithms such as SCC formation.

This section concludes with a brief look at the memory used by the slicers. Data was collected on the space used by each slicer. The two main constituents are the memory needed to store vertices and the memory needed to store edges. While the techniques affect the vertex size, the size of an edge remains constant, at 4 bytes throughout the experiments. Thus, for the most part, the memory usage tracks the vertex size, dampened by the storage used for edges.

Most of the subject programs require negligible memory. For programs on which Std used at least 15Mb of memory, Figure 19 shows the memory used by Std and the most memory efficient slicer +Tran. The +Tran data is divided into the two phases

		Intraprocedural SCCs				Combined Interprocedural SCCs			
Program	Vertices	Vertices in SCCs	SCC count	Vertices per SCC	Percent in SCCs	Vertices in SCCs	SCC count	Vertices per SCC	Percent in SCCs
a2ps	417,513	118,167	2,175	54.3	28.3%	126,622	1,793	70.6	30.3%
acct-6.3	18,531	4,154	116	35.8	22.4%	4,154	116	35.8	22.4%
barcode	10,916	961	117	8.2	8.8%	961	117	8.2	8.8%
bc	14,903	1,779	102	17.4	11.9%	2,080	102	20.4	14.0%
byacc	34,019	7,055	375	18.8	20.7%	7,055	375	18.8	20.7%
cadp	42,212	3,157	149	21.2	7.5%	3,157	149	21.2	7.5%
compress	5,211	350	39	9.0	6.7%	350	39	9.0	6.7%
copia	43,513	100	29	3.4	0.2%	100	29	3.4	0.2%
csurf-pkgs	295,220	17,585	2,976	5.9	6.0%	33,662	2,016	16.7	11.4%
ctags	120,014	28,373	1374	20.6	23.6%	28,373	1,359	20.9	23.6%
cvs	6,219,909	1,016,205	28,769	35.3	16.3%	522,101	786	664.3	8.4%
diffutils	36,079	8,455	335	25.2	23.4%	8,457	335	25.2	23.4%
ed	44,387	17,655	602	29.3	39.8%	17,850	584	30.6	40.2%
empire	496,033	165,822	2,783	59.6	33.4%	165,913	2,756	60.2	33.4%
epwic-1	22,217	4,115	561	7.3	18.5%	4,115	561	7.3	18.5%
espresso	93,326	18,375	929	19.8	19.7%	20,009	983	20.4	21.4%
findutils	30,322	6,208	202	30.7	20.5%	6,430	214	30.0	21.2%
flex2-4-7	39,037	7,327	296	24.8	18.8%	7,366	298	24.7	18.9%
flex2-5-4	55,019	7,345	381	19.3	13.3%	10,916	379	28.8	19.8%
ftpd	56,981	14,395	734	19.6	25.3%	14,454	699	20.7	25.4%
gcc.cpp	19,095	4,151	201	20.7	21.7%	4,430	177	25.0	23.2%
gnubg-0.0	25,138	2,781	323	8.6	11.1%	2,866	328	8.7	11.4%
gnuchess	44,957	10,470	695	15.1	23.3%	10,639	695	15.3	23.7%
gnugo	278,766	27,038	1,693	16.0	9.7%	35,149	1,660	21.2	12.6%
go	111,246	26,052	1,315	19.8	23.4%	26,052	1,315	19.8	23.4%
ijpeg	62,698	12,690	698	18.2	20.2%	17,541	509	34.5	28.0%
indent	25,737	12,229	132	92.6	47.5%	12,249	132	92.8	47.6%
li	889,260	91,302	926	98.6	10.3%	358,953	101	3554.0	40.4%
named	1,326,070	266,182	3,967	67.1	20.1%	439,454	1,966	223.5	33.1%
ntpd	204,420	18,814	2,141	8.8	9.2%	34,130	1,695	20.1	16.7%
oracolo2	20,654	767	71	10.8	3.7%	767	71	10.8	3.7%
prepro	20,578	750	71	10.6	3.6%	750	71	10.6	3.6%
replace	1,097	309	15	20.6	28.2%	309	15	20.6	28.2%
sendmail	1,111,054	324,204	1,446	224.2	29.2%	522,101	786	664.3	47.0%
space	20,018	770	72	10.7	3.8%	770	72	10.7	3.8%
spice	1,774,846	380,480	15,533	24.5	21.4%	426,236	8,701	49.0	24.0%
termutils	8,869	996	67	14.9	11.2%	996	67	14.9	11.2%
tile-forth	59,247	4,073	131	31.1	6.9%	4,857	67	72.5	8.2%
time-1.7	4,583	215	10	21.5	4.7%	215	10	21.5	4.7%
userv-0.95.0	70,796	7,347	398	18.5	10.4%	8,669	361	24.0	12.2%
wdiff.0.5	7,213	1,078	39	27.6	14.9%	1,078	39	27.6	14.9%
which	4,666	452	31	14.6	9.7%	452	31	14.6	9.7%
wpst	54,658	8,397	460	18.3	15.4%	8,921	526	17.0	16.3%
sum	14,241,028	2,649,130	73,479			2,901,709	33,085		
average	331,187	61,608	1,709	29.7	16.9%	67,482	769	142.1	19.2%

Fig. 15. SCC Statistics from the slicers +SCC and w/iSCC-c.

	Initial	Pass 1 SCCs				Pass 2 SCCs			
Program	Vertex count	Vertices in iSCCs	SCC count	Vertices per iSCC	Percent in iSCCs	Vertices in iSCCs	SCC count	Vertices per iSCC	Percent in iSCCs
a2ps	417,513	158,006	1,892	83.5	37.8%	139,777	1,856	75.3	33.5%
acct	18,531	4,165	118	35.3	22.5%	4,158	118	35.2	22.4%
barcode	10,916	961	117	8.2	8.8%	961	117	8.2	8.8%
bc	14,903	2,422	104	23.3	16.3%	3,284	102	32.2	22.0%
byacc	34,019	7,088	375	18.9	20.8%	7,058	374	18.9	20.7%
cadp	42,212	4,415	355	12.4	10.5%	3,958	185	21.4	9.4%
compress	5,211	350	39	9.0	6.7%	350	39	9.0	6.7%
copia	43,513	23,640	30	788.0	54.3%	12,916	47	274.8	29.7%
csurf-packages	295,220	100,758	2,234	45.1	34.1%	77,666	2,097	37.0	26.3%
ctags	120,014	32,371	1,424	22.7	27.0%	30,430	1,347	22.6	25.4%
cvs	6,219,909	3,936,362	15338	256.6	63.3%	3,618,536	13736	263.4	58.2%
diffutils	36,079	9,365	335	28.0	26.0%	8,777	335	26.2	24.3%
ed	44,387	19,864	584	34.0	44.8%	21,032	519	40.5	47.4%
empire	496,033	171,443	2,792	61.4	34.6%	168,023	2,734	61.5	33.9%
EPWIC-1	22,217	4,220	567	7.4	19.0%	4,144	570	7.3	18.7%
espresso	93,326	23,758	999	23.8	25.5%	21,636	978	22.1	23.2%
findutils	30,322	8,279	250	33.1	27.3%	7,446	214	34.8	24.6%
flex2-4-7	39,037	7,816	379	20.6	20.0%	7,406	313	23.7	19.0%
flex2-5-4	55,019	12,204	465	26.2	22.2%	11,228	404	27.8	20.4%
ftpd	56,981	15,760	776	20.3	27.7%	14,888	747	19.9	26.1%
gcc.cpp	19,095	7,347	185	39.7	38.5%	5,960	167	35.7	31.2%
gnubg-0.0	25,138	5,238	338	15.5	20.8%	3,977	336	11.8	15.8%
gnuchess	44,957	12,000	699	17.2	26.7%	10,926	695	15.7	24.3%
gnugo	278,766	42,132	1,753	24.0	15.1%	38,193	1,737	22.0	13.7%
go	111,246	26,486	1,314	20.2	23.8%	26,845	1,312	20.5	24.1%
ijpeg	62,698	26,433	502	52.7	42.2%	20,113	491	41.0	32.1%
indent-1.10.0	25,737	12,312	127	96.9	47.8%	12,261	132	92.9	47.6%
li	889,260	644,016	88	7,318.4	72.4%	578,234	80	7,227.9	65.0%
named	1,326,070	714,200	2,029	352.0	53.9%	650,444	1,942	334.9	49.1%
ntpd	204,420	48,467	1,671	29.0	23.7%	42,854	1,617	26.5	21.0%
oracolo2	20,654	767	71	10.8	3.7%	767	71	10.8	3.7%
prepro	20,578	750	71	10.6	3.6%	750	71	10.6	3.6%
replace	1,097	328	16	20.5	29.9%	311	15	20.7	28.4%
sendmail	1,111,054	724,154	788	919.0	65.2%	673,048	740	909.5	60.6%
space	20,018	770	72	10.7	3.8%	770	72	10.7	3.8%
spice	1,774,846	655,716	9,195	71.3	36.9%	580,454	8,769	66.2	32.7%
termutils	8,869	996	67	14.9	11.2%	996	67	14.9	11.2%
tile-forth-2.1	59,247	12,398	72	172.2	20.9%	11,954	54	221.4	20.2%
time-1.7	4,583	215	10	21.5	4.7%	215	10	21.5	4.7%
userv-0.95.0	70,796	19,454	373	52.2	27.5%	17,139	371	46.2	24.2%
wdiff.0.5	7,213	1,089	41	26.6	15.1%	1,082	41	26.4	15.0%
which	4,666	452	31	14.6	9.7%	452	31	14.6	9.7%
wpst	54,658	10,944	651	16.8	20.0%	10,030	545	18.4	18.4%
sum	14,241,028	7,509,911	49,337	10,885.0		6,851,449	46,198	10,282.5	
average	331,187	174,649	1,147	253.1	27.1%	159,336	1,074	239.1	24.7%

Fig. 16. SCC Statistics for iSCC-s.

of the implementation. Phase One computes SCC, Pack, TS and Tran. It then outputs the SCC representatives in topological order, without the edges removed by Tran. Phase one also outputs the remaining vertices (those in SCCs that are not the SCC representative) so that they can be used as slicing criteria. Phase Two reads in the resulting graph and slices it.

These two phases are run in series: thus, the total memory needed is the larger of the two figures. Compared to the 17% average savings for Phase 1 of +Tran over Std, the average reduction for Phase 2 is 66%; thus, a program requiring three times as much memory can be proceeded by Phase 2 of +Tran than by Std. This is practical, even though Phase 1 requires more memory, because Phase 1 takes only a fraction of the total run time: thus, swapping during this phase might be acceptable.

The memory used by the other slicers, and Phase 1 of +Tran, is proportional to that used by Std. The actual proportion varies slightly from program to program depending primarily on the ratio of vertices to edges. In comparison, the memory drop for Phase 2 of +Tran is more dramatic and more varied. It is more dramatic primarily because fields for preprocessing attributes need not be allocated (*e.g.*, fields used in the SCC construction). It is more varied because Tran has varying

	b-slice				Percent Increase	
	time for	SCC Computation Times (seconds)			+SCC to	+SCC to
Program	+Tran	+SCC	w/iSCC-c	w/iSCC-s	w/iSCC-c	w/iSCC-s
tile-forth-2.1	142.4	0.9	8.0	21.7	784%	2313%
userv-0.95.0	141.4	1.1	11.8	26.6	970%	2318%
copia	22.1	0.4	15.0	90.9	4078%	25153%
flex2-5-4	46.8	0.9	16.0	9.1	1597%	872%
gnugo	3145.8	4.8	16.6	27.3	250%	474%
ftpd	157.2	1.7	18.4	13.6	998%	710%
empire	7664.5	9.1	26.3	37.9	188%	314%
ijpeg	193.8	0.6	44.3	20.1	7410%	3312%
ntpd	1452.7	3.2	61.4	67.8	1794%	1992%
indent-1.10.0	14.6	2.6	124.1	49.8	4709%	1829%
csurf-packages	1363.7	2.7	158.5	903.6	5685%	32877%
a2ps	3835.6	859.5	48,268.9	4,669.8	5516%	443%
li	5181.0	17.7	57,164.9	64,879.4	322865%	366450%
spice	425938.0	1,611.0	81,971.7	37,404.7	4988%	2222%
named	100196.1	1,622.2	223,962.9	70,742.9	13706%	4261%
sendmail	53917.1	2,186.7	600,791.7	55,907.2	27375%	2457%
cvs	280298.1	174.05	2,210,013.5	1,596,458.2	1269658%	917141%

Fig. 17. Computation time comparison for the three SCC techniques (only programs for which one of the three took more than 5 seconds are shown).

	Computation Time		
		iSCC-s	Percent
Program	SCC	intra only	Increase
tile-forth-2.1	0.9	1.2	29%
userv-0.95.0	1.1	1.2	7%
copia	0.4	0.5	36%
flex2-5-4	0.9	9.0	852%
gnugo	4.8	7.3	52%
ftpd	1.7	13.4	699%
empire	9.1	14.8	62%
ijpeg	0.6	0.9	51%
ntpd	3.2	10.2	214%
indent-1.10.0	2.6	49.7	1826%
csurf-packages	2.7	3.6	31%
a2ps	859.5	4,347.8	406%
li	17.7	33.7	90%
spice	1,611.0	3,214.2	100%
named	1,622.2	5,119.8	216%
sendmail	2,186.7	4,391.1	101%
cvs	174.05	1,509.2	767%

Fig. 18. The effect of increased vertex size on the intraprocedural SCC computation time (only the longer running programs from Figure 17 are shown).

degrees of success removing redundant edges, which Phase 2 never need allocate.

6. RELATED WORK

While source code analysis begins with the source code, source code is rarely the best representation for an analysis tool to work with. Many alternate representations have proven useful. Graphs (including their degenerate form, trees) make up a majority of these representations. For example, one of the earliest source code analysis tools, the compiler, led to two of the most widely used graphs: the control-flow graph and the call graph [Fischer and LeBlanc 1988].

Compilers were arguably the first, but are not the only, source-code analysis tool to make use of graphs as internal representations. For example, dependence graphs, developed in connection with aggressive optimizing compilers [Ferrante et al. 1987], have been suggested for use in a variety of alternative applications. These include, for example, program slicing [Binkley and Gallagher 1996; Harman and Hierons 2001; Tip 1995; Weiser 1984], whole program documentation [Balmas 2002], and heap variable modeling [Horwitz et al. 1989].

The optimizations presented here are not restricted to dependence graphs and slicing. Cycle elimination, to which SCC formation belongs, is a very general optimization technique. For example, in data flow analysis it is used to decompose the control flow graph for hybrid analysis algorithms [Lee and Ryder 1994]. Within the SCCs the analysis is iterative and elimination-like during propagation between the SCCs. SCC formation is essential for fast pointer analysis [Fähndrich et al. 1998; Rountev and Chandra 2000; Heintze and Tardieu 2001].

Another example is the efficient computation of path conditions [Snelting et al. 2006], which compute necessary conditions for information flow between two program points. The generation makes heavy use of interval analysis to identify nested SCCs and paths are then decomposed into parts that can be analyzed independently.

In software re-engineering SCC formation can be used as a clustering technique. Koschke [Koschke 1999] uses it for incremental semi-automated component recovery; components in a SCC mutually depend on one another. Systems that analyze dependences and relationships between components, *e.g.*, Rigi [Müller and Klashinsky 1988], use SCC formation to transform cyclic dependence graphs into acyclic graphs.

The present paper is an extended version of a paper presented at the 3rd Source Code Analysis and Manipulation Workshop (SCAM 2003) [Binkley and Harman 2003b]. Also an analysis similar to the one presented here was undertaken by Krinke, who proposed several techniques similar to those discussed in Section 3 for reducing the cost of graph based operations [Krinke 2003]. Two are briefly mentioned herein. First, Krinke observes that the standard computation of SDGs leads to a set of redundant vertices and edges. For example, function parameters that are never used before being defined generate formal-in vertices with no outgoing edges. Such vertices can be removed together with the incident edges without loss of precision, leading to 9% fewer vertices and 13% fewer edges.

Second, his notion of *folding cycles* is essentially the formation of intraprocedural SCCs. He shows that the computation of SCCs for chopping [Jackson and Rollins

		+Tran	
		Phase	
program	Std	1	2
`espresso`	15	10	4
`go`	16	11	4
`ctags`	18	13	5
`ntpd`	22	17	7
`gnugo`	34	27	11
`csurf-pkgs`	35	28	12
`empire`	59	50	17
`a2ps`	86	77	15
`li`	105	90	40
`named`	278	257	121
`sendmail`	384	364	180
`spice`	386	360	208
`cvs`	739	656	271
Average of shown	167	151	69
Average of all	56	48	22

Fig. 19. Slicer memory usage (in MB) for high memory using programs.

1994] must be handled differently to the computation for slicing. The computation of SCCs for slicing includes summary edges, and thus it may fold vertices of different call sites into one SCC. This cannot be used for chopping, because the computation of chops relies upon the mapping of summary edges to call sites. This leads to a smaller reduction than for slicing (a 41% reduction in vertices for slicing and only 26% for chopping).

Krinke also observes that folding interprocedural cycles loses context sensitivity, but achieves a dramatic reduction in graph size (a 66% reduction in vertices and a 76% reduction in edges). The loss of context sensitivity causes not only an avalanche-like loss of precision, but also a large *increase* in average slice construction time. This observation, that waiving context sensitivity typically increases average slice time, has also been reported, with similar results, by Binkley and Harman [Binkley and Harman 2003a] and Krinke [Krinke 2002].

7. SUMMARY

This paper presents five optimization techniques designed to improve graph processing time for interprocedural program slicing. It empirically quantifies the efficiency improvements expected from these different algorithms by studying their effect upon interprocedural slicing performance for a code base of just over one million lines of code. The paper also investigates the effect of compact data representation on the implementation of interprocedural slicing.

The paper also explores the point at which these algorithmic and implementation techniques provide a payoff, relative to the costs of data structure creation. Over all programs studied, only 1.7% of the slices need be computed before the break-even point is reached, indicating that the optimizations are useful for massive slicing, where a much larger percentage of slices would need to be computed. However, the results also indicate, unsurprisingly, that for programs with only a few slices, the break-even point is uneconomical, indicating that the techniques are only worth applying for programs with a large number of slices.

This work is important because many client applications of program slicing require a form of massive slicing–slicing on many or all of the possible criteria within the program. The number of slicing criteria in a program is of a similar order to the number of lines of code in the program, so efficiency of slice computation time is an important issue for massive slicing of large programs.

The empirical evidence provided by the studies reported in the paper gives baseline data for future work. The data are also useful to tool builders concerned with the decision as to which, if any, graph technique to incorporate into a tool. The advantages and disadvantages of identifying strongly-connected components or using topological order, for example, can be better evaluated given the data presented in this paper. Future work will investigate the extent to which the techniques explored here can be extended to apply to other graph-based program analyses.

8. ACKNOWLEDGMENTS

The authors wish to thank GrammaTech Incorporated (`www.grammatech.com`) for providing CodeSurfer. Mark Harman is supported by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two grants from DaimlerChrysler. Dave Binkley is supported by National Science Foundation grant CCR-0305330. Kathy Newstead and Jeri Hanly assisted with the presentation. The authors also wish to thank the anonymous referees. Their detailed, constructive and thoughtful comments greatly improved the presentation of the results in this paper.

REFERENCES

Agrawal, H. and Horgan, J. R. 1990. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, New York). 246–256.

Anderson, P., Binkley, D., Rosay, G., and Teitelbaum, T. 2001. Flow insensitive points-to sets. In *Proceedings of the first IEEE Workshop on Source Code Analysis and Manipulation* (Florence, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 79–89.

Balmas, F. 2002. Using dependence graphs as a support to document programs. In 2^{st} *IEEE International Workshop on Source Code Analysis and Manipulation* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 145–154.

Beszédes, A. and Gyimóthy, T. 2002. Union slices for the approximation of the precise slice. In *IEEE International Conference on Software Maintenance* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 12–20.

Bieman, J. M. and Ott, L. M. 1994. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20,* 8 (Aug.), 644–657.

Binkley, D. and Harman, M. 2005a. Forward slices are smaller than backward slices. In 5^{th} *IEEE International Workshop on Source Code Analysis and Manipulation* (Budapest, Hungary, September 30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 15–24.

Binkley, D. and Harman, M. 2005b. Locating dependence clusters and dependence pollution. In 21^{st} *IEEE International Conference on Software Maintenance* (Budapest, Hungary, September 30th-October 1st 2005). IEEE Computer Society Press, Los Alamitos, California, USA, 177–186.

Binkley, D. W. 1998. The application of program slicing to regression testing. *Information and Software Technology Special Issue on Program Slicing 40,* 11 and 12, 583–594.

Binkley, D. W. and Gallagher, K. B. 1996. Program slicing. In *Advances in Computing, Volume 43*, M. Zelkowitz, Ed. Academic Press, 1–50.

Binkley, D. W. and Harman, M. 2003a. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *IEEE International Conference on Software Maintenance* (Amsterdam, Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 44–53.

Binkley, D. W. and Harman, M. 2003b. Results from a large–scale study of performance optimization techniques for source code analyses based on graph reachability algorithms. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)* (Amsterdam, Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 203–212.

Binkley, D. W. and Harman, M. 2004. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering 30,* 11, 715–735.

Binkley, D. W., Harman, M., Raszewski, L. R., and Smith, C. 2000. An empirical study of amorphous slicing as a program comprehension support tool. In 8^{th} *IEEE International Workshop on Program Comprehension* (Limerick, Ireland). IEEE Computer Society Press, Los Alamitos, California, USA, 161–170.

Binkley, D. W., Horwitz, S., and Reps, T. 1995. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4,* 1, 3–35.

Black, S. E. 2001. Computing ripple effect for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice 13*, 263–279.

Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. 1994. Software salvaging based on conditions. In *International Conference on Software Maintenance* (Victoria, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 424–433.

Canfora, G., Cimitile, A., and Munro, M. 1994. RE^2: Reverse engineering and reuse re-engineering. *Journal of Software Maintenance: Research and Practice 6,* 2, 53–72.

Chilimbi, T., Davidson, B., Larus, J., and Hill, M. 1999. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia). Association for Computer Machinery, New York, 13–24.

Cimitile, A., De Lucia, A., and Munro, M. 1995a. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance* (Nice, France). IEEE Computer Society Press, Los Alamitos, California, USA, 124–133.

Cimitile, A., De Lucia, A., and Munro, M. 1995b. Qualifying reusable functions using symbolic execution. In *Proceedings of the* 2^{nd} *Working Conference on Reverse Engineering* (Toronto, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 178–187.

Danicic, S., De Lucia, A., and Harman, M. 2004. Building executable union slices using conditioned slicing. In 12^{th} *International Workshop on Program Comprehension* (Bari, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 89–97.

Eisenbarth, T., Koschke, R., and Vogel, G. 2002. Static trace extraction. In *IEEE Working Conference on Reverse Engineering* (Richmond, Virginia, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 128–137.

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

Fähndrich, M., Foster, J. S., Su, Z., and Aiken, A. 1998. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montréal, Canada). Association for Computer Machinery, 85–96.

Fischer, C. N. and LeBlanc, R. J. 1988. *Crafting a Compiler.* Benjamin/Cummings Series in Computer Science. Benjamin/Cummings Publishing Company, Menlo Park, CA.

Fox, C., Harman, M., Hierons, R. M., and Danicic, S. 2001. Backward conditioning: a new program specialisation technique and its application to program comprehension. In 9^{th} *IEEE International Workshop on Program Comprenhesion* (Toronto, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 89–97.

Gallagher, K. B. and Lyle, J. R. 1991. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17,* 8 (Aug.), 751–761.

Harman, M. and Danicic, S. 1995. Using program slicing to simplify testing. *Software Testing, Verification and Reliability 5,* 3 (Sept.), 143–162.

Harman, M. and Hierons, R. M. 2001. An overview of program slicing. *Software Focus 2,* 3, 85–92.

Harman, M., Hierons, R. M., Danicic, S., Howroyd, J., Laurence, M., and Fox, C. 2001. Node coarsening calculi for program slicing. In 8th *Working Conference on Reverse Engineering* (Stuttgart). IEEE Computer Society Press, Los Alamitos, California, USA, 25–34.

Heintze, N. and Tardieu, O. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, C. Norris and J. J. B. Fenwick, Eds. ACM SIGPLAN Notices, vol. 36.5. ACMPress, New York, 254–263.

Hierons, R. M., Harman, M., and Danicic, S. 1999. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability 9,* 4, 233–262.

Hierons, R. M., Harman, M., Fox, C., Ouarbya, L., and Daoudi, M. 2002. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability 12*, 23–28.

Horwitz, S., Pfeiffer, P., and Reps, T. 1989. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation* (Portland, OR, USA). ACM SIGPLan Notices, 28–40.

Horwitz, S., Prins, J., and Reps, T. 1989. Integrating non–interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11,* 3 (July), 345–387.

Horwitz, S., Reps, T., and Binkley, D. W. 1988. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* Atlanta, Georgia, 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

Horwitz, S., Reps, T., and Binkley, D. W. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12,* 1, 26–61.

Jackson, D. and Rollins, E. J. 1994. Chopping: A generalisation of slicing. Tech. Rep. CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. July.

Koschke, R. 1999. An incremental semi-automatic method for component recovery. In *Proceedings: Sixth Working Conference on Reverse Engineering* (Atlanta, Georgia USA). IEEE Computer Society Press, Los Alamitos, California, USA, 256–269.

Krinke, J. 2002. Evaluating context-sensitive slicing and chopping. In *IEEE International Conference on Software Maintenance* (Montreal, Canada). IEEE Computer Society Press, Los Alamitos, California, USA, 22–31.

Krinke, J. 2003. Advanced slicing of sequential and concurrent programs. Ph.D. thesis, Universität Passau.

Lee, Y. and Ryder, B. G. 1994. Effectively exploiting parallelism in data flow analysis. *The Journal of Supercomputing 8,* 3 (Nov.), 233–262.

Liang, D. and Harrold, M. J. 1999. Efficient points-to analysis for whole-program analysis. In *ESEC/FSE '99*, O. Nierstrasz and M. Lemoine, Eds. Lecture Notes in Computer Science, vol. 1687. Springer-Verlag / ACM Press, 199–215.

Longworth, H. D., Ott, L. M., and Smith, M. R. 1986. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86).* 383–389.

Meyers, T. and Binkley, D. W. 2004. Slice-based cohesion metrics and software intervention. In 11th *IEEE Working Conference on Reverse Engineering* (Delft University of Technology, the Netherlands). IEEE Computer Society Press, Los Alamitos, California, USA, 256–266.

Müller, H. A. and Klashinsky, K. 1988. Rigi—A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering.* IEEE Computer Society Press, Singapore. April 11-15, 1988, 80–86.

Mock, M., Atkinson, D. C., Chambers, C., and Eggers, S. J. 2002. Improving program slicing with dynamic points-to data. In *Proceedings of the* 10th *ACM SIGSOFT Symposium on the*

Foundations of Software Engineering (FSE-02), W. G. Griswold, Ed. ACM Press, New York, 71–80.

Ott, L. M. 1992. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10th *Annual Software Reliability Symposium.* 16–23.

Ott, L. M. and Bieman, J. M. 1992. Effects of software changes on module cohesion. In *IEEE Conference on Software Maintenance.* 345–353.

Ott, L. M. and Thuss, J. J. 1989. The relationship between slices and module cohesion. In *Proceedings of the* 11th *ACM Conference on Software Engineering.* 198–204.

Ott, L. M. and Thuss, J. J. 1993. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (Baltimore, Maryland, USA). IEEE Computer Society Press, Los Alamitos, California, USA, 71–81.

Ottenstein, K. J. and Ottenstein, L. M. 1984. The program dependence graph in software development environments. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environmt, SIGPLAN Notices 19,* 5, 177–184.

Podgurski, A. and Clarke, L. 1990. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering 16,* 9, 965–79.

Reps, T. 1998. Program analysis via graph reachability. *Information and Software Technology Special Issue on Program Slicing 40,* 11 and 12, 701–726.

Reps, T., Horwitz, S., Sagiv, M., and Rosay, G. 1994. Speeding up slicing. In *ACM Foundations of Software Engineering* (New Orleans, LA). ACM SIGSOFT Software Engineering Notes 19, 5 (December 1994), 11–20.

Rilling, J., A.Seffah, and J.Lukas. 2001. MOOSE – a software comprehension framework. In 5th *World Multi-Conference on Systemics, Cybernetics and Informatics* (Orlando, Florida). 312–318.

Rountev, A. and Chandra, S. 2000. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* ACM Sigplan Notices, vol. 35.5. ACM Press, New York, 47–56.

Snelting, G., Robschink, T., and Krinke, J. 2006. Efficient path conditions in dependence graphs for software safety analysis. *ACM Transactions on Software Engineering and Methodology.* To appear.

Tip, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 3,* 3 (Sept.), 121–189.

Weiser, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, University of Michigan, Ann Arbor, MI.

Weiser, M. 1981. Program slicing. In 5th *International Conference on Software Engineering.* San Diego, CA, 439–449.

Weiser, M. 1982. Programmers use slices when debugging. *Communications of the ACM 25,* 7 (July), 446–452.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering 10,* 4, 352–357.

Zhao, J. 2002. Slicing aspect-oriented software. In 10th *IEEE International Workshop on Program Comprehension* (Paris, France). IEEE Computer Society Press, Los Alamitos, California, USA, 351–260.