Results from a Large–Scale Study of Performance Optimization Techniques for Source Code Analyses Based on Graph Reachability Algorithms

David Binkley
Loyola College
Baltimore MD
21210-2699, USA
binkley@cs.loyola.edu

Mark Harman
Brunel University
Uxbridge, Middlesex
UB8 3PH, UK
Mark.Harman@brunel.ac.uk

Keywords: Slicing, Internal Representation, Performance Enhancement

Abstract

Internally, many source code analysis tools make use of graphs. For example, one of the oldest and most widely used internal graphs is the control-flow graph developed for use within a compiler. Work on compilation has also led to the development of the call graph, the procedure dependence graph (PDG), and the static-single assignment (SSA) graph. Compilers are not the only source-code analysis tools to use internal graphs. A variety of software engineering tools incorporate a variety of different graphs.

A study of techniques that improve graph–based program analysis is presented. Several different techniques are considered, including forming strongly-connected components, topological sorting, and removing transitive edges. Graph reachability, a pervasive graph analysis operation, is used as a representative graph analysis operation in the study.

Data collected from a test bed of just over 1,000,000 lines of code is presented. This data illustrates the impact on computation time of the improvement techniques. Overall, the combination of all techniques produces a 71% reduction in run-time (and 64% reduction in memory usage). In other words, they increase the size of the problem that can be effectively handled by factor of over three times.

1. Introduction

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, lead to two of the widest used graphs: the control-flow graph and the call graph [8].

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 [7], have been suggested for use in a variety of alternative applications. These include program slicing [4, 10, 17, 18], whole program documentation [3], heap variable modeling [12].

A common operation performed by tools is *graph reachability*. For example, the question "Can procedure P (transitively) call procedure Q?" is a simple graph reachability question on the program's call-graph. Similarly, the question "Does the value assigned to `x` at statement `s` reach a use of `x` at statement `t`" begins with the reachability question "is there a path from `s` to `t` in the program's control-flow graph?".

This paper reports the results of experiments with a collection of techniques whose goal is to improve source code analysis tools that employ graphs as internal representations. Data was collected from a test bed of just over 1,000,000 lines of source code. The results quantify the impact on the run-time (and thus applicability) of the techniques. Such data assists tool designers when making decisions regarding the incorporation of a particular graph or technique.

For this work to achieve good external validity [19], it is necessary to study a graph operation that strikes a balance between complexity and understandability. Too simple an operation and the analysis task becomes trivial and thereby fails to discriminate between the techniques. On the other hand, the effort to understand too complex an operation can obscure results obtained using the techniques.

This study uses precise interprocedural slicing of dependence graphs (hereafter *slicing*) as a representative graph operation [13, 18]. Interprocedural slicing is neither too simple nor too complex an operation. Rather, as explained in Section 3, it strikes the appropriate balance.

This paper focuses on graphs used as internal representations. Tools also use graphs as part of an external (display) engine when various connections between program components are illustrated (for example, Ball's work [2], and Ringling and Mudur's work [16]). A large variety of external graphs trace their original to object orientation and, in particular, to the UML [9].

The remainder of the paper is organized as follows: Section 2 provides background on program slicing and the tools used to collect the data. Section 3 describes the optimization techniques, while Section 4 reports the results of the study. Finally, Section 5 summarizes the paper.

2. Background

This section describes the interprocedural slicing algorithm, the subject programs studied, and the data collection environment.

2.1. 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 [13]. 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 C's value-result 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 (or 'flow') dependence edges. A control dependence edge is labeled either **true** or **false**. It connects a predicate vertex v to a vertex u if, during execution, whenever 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). The only kind of data-dependence edge used is the *flow dependence edge*. A 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. The dependence edges are safe approximations to the semantic dependences found in the program [15], which are generally not computable and therefore must be (safely) approximated.

Taken with respect to vertex v, a slice includes the vertices that potentially affect the computation represented at v. 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. A two-pass reachability algorithm [13] avoids this imprecision and also provides a more interesting algorithm for study in Sections 3 and 4.

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. For the slice taken with respect to vertex v, the the two pass algorithm first identifies the necessary vertices in v's procedure P and 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 which call site to "return" to. The second pass identifies the necessary vertices from called procedures by only descending into procedures, again side-stepping the calling-context problem.

While a simple reachability can be implemented by a recursive algorithm that marks vertices as it encounters them, the two pass reachability causes certain vertices to be revisited. As a result the placement of vertices in memory has greater importance.

2.2. 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 consist of publicly available programs. Figure 1 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 [1, 6, 14]. Second, a slice was computed for every vertex that represents executable source code. It turns out that 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–esc 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. Considering 43 subject programs provides a measure of diversity to make possible general conclusions about the techniques. This diversity is useful in improving confidence in the external validity of the techniques applied in Section 4.

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

Figure 1. Subjects

2.3. 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 for the shared memory bus slows each down by 15.5% relative to a single processor in an identical environment. CPU utilization was $99^{+}\%$ for all runs of the slicer. Each program 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 SDG includes vertices that do not directly represent a particular source statement. The most common cause of these 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`.

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

Figure 2. Five Increasingly Optimized Versions of the Slicer

3. Techniques

This section introduces the four optimization techniques investigated. The next section reports their impact on the time and space taken by the slicer. As described in Figure 2, the four techniques lead to five versions of the slicer.

To begin with, Std is the standard (unoptimized) version used to generate base-line data. It represents a straightforward implementation of the two pass interprocedural slicing algorithm as presented by Horwitz et al. [13]. The vertex data structure and excerpts from the key part of the slicing algorithm are shown in Figure 3.

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 only considering 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 these vertices to mark the vertices of Pass 2. Function `b2_flood` is similar to `b1_flood` except *actual-out* edges into called procedure are traversed rather than *actual-in* and *call* edges out to calling procedures.

Four techniques are used to produce the four optimized versions of the Std slicer as described in Figure 2. The techniques include two algorithmic techniques: the formation of strongly connected components, SCC and topological sorting, +TS, and two "low level" techniques: vertex size reduction, +Pack and redundant transitive edge removal, +Tran.

Each technique is incorporated to create a progressively more optimized version of the slicer. The SCC optimization is used to augment the unoptimized slicer, Std. The lower level packing algorithm augments SCC to give +Pack. This is further augmented by topological sorting to give +TS. Finally, transitive edge removal is added, to produce the fully optimized version, +Tran.

The first technique forms SCCs. The two key insights here are that all vertices in an SCC will have the same slice and, any slice that includes a vertex from an SCC, by definition, must include all the vertices of the SCC. Thus, SCCs can be collapsed into a single representative node. This is an instance of a minimal, consistent and sharp node coarsening [11], which means that the slices produces are *identical* to those which would have been produced without merging together nodes into SCCs. SCC includes a strait-forward implementation of SCC construction, which, as seen in Figure 4, increases the size of a vertex. SCC formation is a preprocessing step. The core slicing algorithm is unaffected by this change.

The computation of SCCs underscores the appropriateness of interprocedural slicing as a target operation for the study. Using a simpler operation, such as intraprocedural slicing, SCCs could have been formed over the entire graph. Identifying such SCCs would reduce the graph to a DAG and thus allow a DAG–specific algorithm to be applied. Allowing this new class of algorithms to be applied would have reduced the external validity of the experiment. However, in order to preserve the precision of an interprocedural slice, SCCs do not cross procedure boundaries. Thus, only intraprocedural SCCs are formed by the algorithm. This ensures that the problem retains all the characteristics as a graph traversal algorithm that it had before SCC formation.

+Pack incorporates data structure changes that reduce the size of a vertex and reorder the structure fields to exploit cache behavior. 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. In comparison, a vertex from the +Pack version takes only 24. This reduction was achieved by re-using 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 another field, the `low_link` field (see Figure 4) during SCC formation.

```
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);
  }
}
```

Figure 3. Std**: The Unoptimized Version; Source Excerpt**

```
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;
```

Figure 4. SCC**: Vertex Declaration**

The second change employs bit packing to collapse the four edge pointers of the SCC version (and Std version) to two (as shown in Figures 4 and 5). 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" can be packed into these bits. The core of the function `b1_flood` in the +Pack version is

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

This classic time-space tradeoff produces extra work at runtime in the from 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.

The +Pack version also pays attention to 2 memory layout issues. First, related fields of a vertex structure are made adjacent in the hope that they would be in the same cache line and thus more accessible. Second, vertices' incoming and outgoing edges were allocated from separate heaps; thus, providing better locality of reference.

The third optimization technique, includes the second algorithmic change. It considers the order of the vertices in memory. 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 read in. However, topological sorting proved beneficial and is incorporated into the +TS version. Note that SCC formation is a necessary preprocessing step to topological sorting, as the sort requires a

```
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;
```

Figure 5. +Pack**: Vertex Declaration**

DAG, which is why the optimization techniques are considered in the order embodied in the versions of the slicer.

Here again interprocedural slicing provides a good choice for the operation to study. SCC formation has transformed each PDG into a DAG. However cycles involving interprocedural edges remain. Thus, each procedure is output separately in topological order.

The slicer was divided into two processes to facilitate node rearrangement. This also allows a further reduction in vertex size as fields used only during SCC formation are no longer needed.

Finally, the +Tran version (which includes all optimizations considered in this paper) incorporates a reachability specific transformation. It removes transitive edges. For example, a code sequences such as

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

includes a "redundant" flow dependence edge: There are flow 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.

At one extreme, edges could be added to transitively close the graph; thus reducing reachability to adjacency (*i.e.*, connected by an edge). Adding these edges should speed the computation of reachability (at a tremendous space cost). The other extreme is to use the least graph whose transitive closure is the same as the transitive closure of the input graph. The removal of these "extraneous transitive" edges should, in principal, have the opposite effect: It should require less memory at the expense of increased computation time.

4. Analysis

This section describes the results of running each version of the slicer from Section 3 on the programs described in Section 2. Summary data covering all executions are first discussed, followed by a more detailed examination of each version of the slicer. This section also presents some statistics collected while running the various techniques (such as the number and size of SCCs).

Relative computation times for all five versions are shown in Figure 6. This same data is should graphically in Figure 8. The times are scaled so that the run-time for the unoptimized version (Std) is 1.00 for each program. This was done because the raw computation times for this version ranged from 0.05 seconds to 768,391 seconds, which makes summary statistics such as averages meaningless (the un-scaled times for the +Tran version, which includes all optimizations, are shown in Figure 9). In Figure 6, the last three rows show the sum, average, and the cumulative runtime reduction. As can be seen in the data, the average speedup increases with each technique incorporated into the slicer. Combining all four techniques produces an average 71% time reduction.

The remainder of this section considers each version independently. To begin with, Figure 7 shows statistics related to SCC formation. This includes the number of SCCs, the size of SCCs, and the percentage of the vertices in SCCs. This latter value ranges from a low of 0.2% to a high of 47.3% 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 speedup. Programs `a2ps` and `diffutils` with speedups 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 is multiplied by repeat occurrence.

Another interesting thing seen in the data from the Column for the SCC version of Figure 6 is that SCC formation was not beneficial for analysis times of all programs. The worst case, `csurf-pkgs`, has a "speed-up" of 0.82 (*i.e.*, runs 18% slower). Lack of a universal benefit is caused by two conflicting affects on run-time. First, an increase in vertex size was required to maintain data used by SCC formation. This size increase tends to increase computation time as it negatively affects cache performance. On the other

name	Version				
	Std	SCC	+Pack	+TS	+Tran
a2ps	1.00	3.33	4.59	6.64	7.13
acct-6.3	1.00	1.54	2.74	4.35	5.70
barcode	1.00	0.99	2.15	3.62	5.51
bc	1.00	1.06	1.84	3.10	3.88
byacc	1.00	1.38	2.08	3.28	3.58
cadp	1.00	1.14	1.63	2.94	3.28
compress	1.00	1.25	1.67	1.67	1.67
copia	1.00	0.88	1.18	1.43	1.50
csurf-pkgs	1.00	0.82	1.11	1.39	1.47
ctags	1.00	1.17	1.60	2.36	2.57
cvs	1.00	0.86	1.07	1.56	1.73
diffutils	1.00	1.80	2.68	3.93	4.43
ed	1.00	1.83	2.62	3.72	4.20
empire	1.00	1.32	1.82	2.57	2.79
epwic-1	1.00	1.14	2.32	5.32	7.36
espresso	1.00	1.18	1.59	2.25	2.48
findutils	1.00	1.56	2.48	3.94	4.56
flex2-4-7	1.00	1.63	2.29	3.16	3.37
flex2-5-4	1.00	2.34	3.23	4.66	5.29
ftpd	1.00	1.79	2.44	3.24	3.44
gcc.cpp	1.00	1.48	2.37	3.89	4.62
gnubg-0.0	1.00	1.00	1.52	2.31	2.64
gnuchess	1.00	1.34	1.91	2.79	3.25
gnugo	1.00	0.93	1.32	1.73	1.84
go	1.00	1.13	1.87	2.37	2.63
ijpeg	1.00	1.18	1.79	2.22	2.57
indent	1.00	5.40	7.85	11.59	14.01
li	1.00	0.90	1.20	1.62	1.77
named	1.00	1.35	1.66	2.16	2.25
ntpd	1.00	1.10	1.49	1.90	2.13
oracolo2	1.00	0.87	1.43	2.19	2.51
prepro	1.00	0.89	1.43	2.44	2.65
replace	1.00	1.67	2.50	2.50	2.50
sendmail	1.00	1.67	1.82	2.33	2.36
space	1.00	0.89	1.50	2.38	2.69
spice	1.00	1.02	1.52	1.80	1.80
termutils	1.00	1.09	3.15	4.31	5.28
tile-forth	1.00	1.00	1.41	1.80	1.97
time-1.7	1.00	1.25	2.50	1.67	1.67
userv-0.95.0	1.00	1.02	1.44	1.91	2.06
wdiff.0.5	1.00	1.56	1.87	1.87	2.33
which	1.00	1.33	1.50	1.71	2.00
wpst	1.00	1.19	1.72	2.45	2.75
sum	43.00	60.28	89.91	127.07	146.24
average	1.00	1.40	2.09	2.96	3.40
cummulative reduction		29%	52%	66%	71%

Figure 6. Speedup (scaled to Std = 1.00)

Program	Vertices	SCC count	Vertices in SCCs	Vertices per SCC	Percent in SCCs
a2ps	417,513	2,175	118,167	54.3	28.3%
acct-6.3	18,531	116	4,154	35.8	22.4%
barcode	10,916	117	961	8.2	8.8%
bc	14,903	102	1,779	17.4	11.9%
byacc	34,019	375	7,055	18.8	20.7%
cadp	42,212	149	3,157	21.2	7.5%
compress	5,211	39	350	9.0	6.7%
copia	43,513	29	100	3.4	0.2%
csurf-pkgs	295,220	2,976	17,585	5.9	6.0%
ctags	120,014	1,187	31,234	26.3	26.0%
cvs	6,219,909	28,769	1,016,205	35.3	16.3%
diffutils	36,079	335	8,455	25.2	23.4%
ed	44,387	602	17,655	29.3	39.8%
empire	496,033	2,783	165,822	59.6	33.4%
epwic-1	22,217	561	4,115	7.3	18.5%
espresso	93,326	929	18,375	19.8	19.7%
findutils	30,322	202	6,208	30.7	20.5%
flex2-4-7	39,037	296	7,327	24.8	18.8%
flex2-5-4	55,019	381	7,345	19.3	13.3%
ftpd	56,981	734	14,395	19.6	25.3%
gcc.cpp	19,095	201	4,151	20.7	21.7%
gnubg-0.0	25,138	323	2,781	8.6	11.1%
gnuchess	44,957	695	10,470	15.1	23.3%
gnugo	278,766	1,693	27,038	16.0	9.7%
go	111,246	1,315	26,052	19.8	23.4%
ijpeg	62,698	698	12,690	18.2	20.2%
indent	25,737	132	12,229	92.6	47.5%
li	889,260	926	91,302	98.6	10.3%
named	1,326,070	3,967	266,182	67.1	20.1%
ntpd	204,420	2,141	18,814	8.8	9.2%
oracolo2	20,654	71	767	10.8	3.7%
prepro	20,578	71	750	10.6	3.6%
replace	1,097	15	309	20.6	28.2%
sendmail	1,111,054	1,446	324,204	224.2	29.2%
space	20,018	72	770	10.7	3.8%
spice	1,774,846	15,533	380,480	24.5	21.4%
termutils	8,869	67	996	14.9	11.2%
tile-forth	59,247	131	4,073	31.1	6.9%
time-1.7	4,583	10	215	21.5	4.7%
userv-0.95.0	70,796	398	7,347	18.5	10.4%
wdiff.0.5	7,213	39	1,078	27.6	14.9%
which	4,666	31	452	14.6	9.7%
wpst	54,658	460	8,397	18.3	15.4%
sum	14,241,028	73,292	2,651,991		
average	331,187	1,704	61,674	29.9	16.9%

Figure 7. SCC Statictics

hand, not repeatedly processing the vertices that make up each SCC tends to decrease computation time. Over all programs studied the formation of SCCs causes a net speedup, as seen in the last three rows of the table.

Next, consider the +Pack column of Figure 6, which shows the effect of the two related packing techniques. First, the vertex size was reduced and second structure fields were reordered to exploit cache layout effects. In comparison with SCC (the previous version), size reduction accounts for three quarters of the improvement and layout the remaining quarter.

To isolate the effect of SCC formation requires forming 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); thus, obtaining the true effect of SCC formation. Doing so, the average speedup is 1.92 (compared to 2.09 when including the reordering change). More importantly, each of the 43 programs shows some reduction. Thus, given constant vertex size, SCC formation always leads to an improvement.

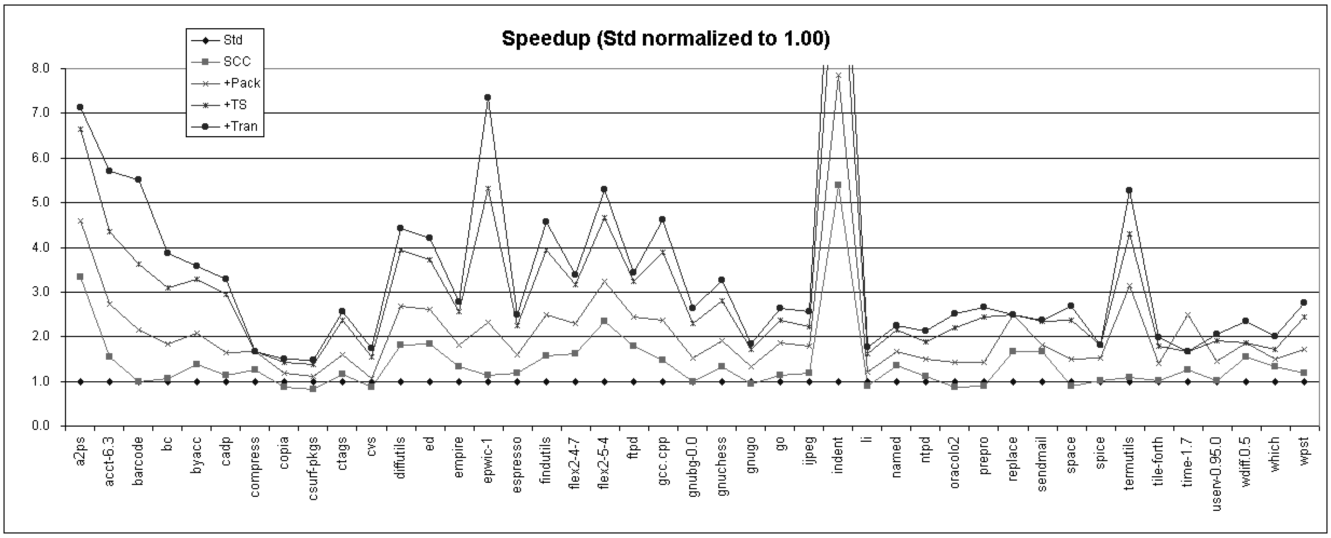

Figure 8. Speedup for all five slicer versions on all programs

For +Pack, the smallest speedup is 1.11 and the largest is 7.42. 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 programs and give general guidelines for program development.

Next, consider the column for top sorting (the +TS version), which provides another significant improvement. This improvement has two main constituents: the top sorting and a drop in vertex size from 20 to 16 bytes. While not shown in the figure, top sorting accounts for 86% of the reduction. This result suggests future work for compiler writers and tools builders on techniques for automatically laying out data structures. Such work would mirror similar lower-level layout work [5].

The only slowdown with this version is for program `time-1.7` whose "speedup" goes from 2.50 to 1.67. `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.

Finally, +Tran incorporates transitive edge removal, which at first glance 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. In essence this means the traversing such edges is wasted effort.

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

For this version of the slicer, two additional views of the data are presented. First, the following table shows the time taken (as a percentage) by each step in the computation. Note that slicing includes the time to compute both backward and forward slices [13] for every statement in the program. The results for forward slices are comparable to those for backwards slices. 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%

Program	Average Slice Size (vertices)	Total Time (sec)	Pace (KLOC/sec)
a2ps	237,041	3,835.59	463
acct-6.3	1,770	0.97	6,783
barcode	4,010	1.47	5,315
bc	8,986	7.29	6,169
byacc	9,731	16.75	1,098
cadp	4,738	13.42	1,451
compress	655	0.06	2,805
copia	13,034	22.08	74
csurf-pkgs	67,158	1,363.73	261
ctags	72,960	360.34	581
cvs	4,872,236	280,298.17	27
diffutils	9,477	28.63	2,802
ed	30,883	85.59	1,662
empire	336,665	7,664.52	505
epwic-1	2,669	2.45	5,267
espresso	40,583	351.29	793
findutils	10,160	25.21	3,215
flex2-4-7	12,510	30.69	1,756
flex2-5-4	9,971	46.76	1,108
ftpd	27,788	157.22	1,225
gcc.cpp	10,097	12.47	1,272
gnubg-0.0	9,520	20.88	1,239
gnuchess	22,385	73.50	1,701
gnugo	145,428	3,145.85	147
go	92,147	785.72	1,071
ijpeg	28,414	193.84	1,390
indent	14,136	14.64	1,511
li	684,492	5,181.00	14
named	987,310	100,196.17	81
ntpd	98,652	1,452.69	600
oracolo2	6,363	14.38	3,329
prepro	6,243	13.50	3,471
replace	423	0.02	9,417
sendmail	820,749	53,917.14	48
space	6,494	13.63	2,240
spice	849,030	425,938.17	36
termutils	2,126	0.40	12,424
tile-forth	29,929	142.44	157
time-1.7	324	0.03	14,922
userv-0.95.0	30,217	141.41	270
wdiff.0.5	775	0.12	12,916
which	811	0.06	16,337
wpst	10,151	50.52	1,316
sum	9,629,238	885,620.81	129,267
aveage	223,936	20,586.94	3,006

Figure 9. Computation Pace

The final data presented for +Tran considers the raw slicing times. Figure 9 includes the average slice size, the total time taken to compute the two-pass closure for every vertex that represents executable code 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 time; just the actual computation of the closure.

Total time (shown in Column 3) was used to generate a liner 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 closure. 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 shows 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 is a pace of approximately 100 KLOC per second.

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 a pace of 3,006 KLOC per second. This figure is higher than the 100 KLOC because it 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.

5. Summary

Many source code analyses use graphs as internal representations. This paper quantifies the improvement expected from different algorithms designed to improve graph processing time. This is done using the well–known two pass interprocedural slicing algorithm as an example graph operation. These results should generalize to related problems, in particular those involving graph reachability.

The empirical data in Section 4 provides base-line data for future work on source code analysis tools. Proposed algorithms (based on related graph manipulations) should expect similar performance to the traversals analyzed in Section 3 and shown in Figure 9.

The data is also useful to tool builders looking to decide which, if any, graph techniques to incorporate into their tool. The pros and cons of identifying strongly-connected components or using topological order, for example, can be better evaluated given the data presented in this paper.

6. Acknowledgements

The authors wish to thank GrammaTech Incorporated (`http://www.grammatech.com`) for providing CodeSurfer, Gregg Rothermel for providing the `space` program. Mark Harman is supported, in part, by EPSRC Grants GR/R98938, GR/M58719, GR/M78083 and GR/R43150 and by two development grants from DaimlerChrysler. Dave Binkley is supported by National Science Foundation grant CCR-9803665.

References

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

[2] T. Ball and S. G. Eick. Visualizing program slices. In A. L. Ambler and T. D. Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 288–295, Los Alamitos, CA, USA, Oct. 1994. IEEE Computer Society Press.

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

[4] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances of Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[5] T. Chilimbi, B. Davidson, J. Larus, and M. Hill. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, New York, May 1999.

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

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

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

[9] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 2/E*. Addison Wesley Professional, 2999.

[10] M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

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

[12] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1989)*, ACM SIGPLAN Notices*, pages 28–40. ACM SIGPLan Notices, July 1989.

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

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

[15] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(8), 1990.

[16] J. Rilling and S. P. Mudur. On the use of metaballs to visually map source code structures and analysis results onto 3d space. In 10^{th} *Working Conference on Reverse Engineering*, pages 42–52, Richmond, Virginia, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.

[18] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[19] M. Zelkowitz, D. Wallace, and D. Binkley. *Experimental validation of new software technology*, chapter 12. SE Empirical Validation (Natalia Juristo and Ana Moreno eds.). Molex, 2003.