Interprocedural Path Profiling

David Melski and Thomas Reps

Computer Sciences Department, University of Wisconsin
1210 West Dayton Street, Madison, WI, 53706, USA
{melski,reps}@cs.wisc.edu

Abstract. In path profiling, a program is instrumented with code that counts the number of times particular path fragments of the program are executed. This paper extends the *intra*procedural path-profiling technique of Ball and Larus to collect information about *inter*procedural paths (i.e., paths that may cross procedure boundaries).

1 Introduction

In path profiling, a program is instrumented with code that counts the number of times particular finite-length path fragments of the program's control-flow graph — or *observable paths* — are executed. A path profile for a given run of a program consists of a count of how often each observable path was executed. This paper extends the *intra*procedural path-profiling technique of Ball and Larus [3] to collect information about *inter*procedural paths (i.e., paths that may cross procedure boundaries).

Interprocedural path profiling is complicated by the need to account for a procedure's calling context. There are really two issues:

- *What is meant by a procedure's "calling context"?* Previous work by Ammons et al. [1] investigated a hybrid intra-/interprocedural scheme that collects separate *intra*procedural profiles for a procedure's different calling contexts. In their work, the "calling context" of procedure P consists of the *sequence of call sites* pending on entry to P. In general, the sequence of pending call sites is an abstraction of any of the paths ending at the call on P.
  The path-profiling technique presented in this paper profiles true *inter*procedural paths, which may include call and return edges between procedures, paths through pending procedures, and paths through procedures that were called in the past and completed execution. This means that, in general, our technique maintains finer distinctions than those maintained by the profiling technique of Ammons et al.
- *How does the calling-context problem impact the profiling machinery?* In the method presented in this paper, the "naming" of paths is carried out via an edge-labeling scheme that is in much the same spirit as the path-naming scheme of the Ball-Larus technique, where each edge is labeled with a number, and the "name" of a path is the sum of the numbers on the path's edges. However, to handle the calling-context problem, in our method

S. Jähnichen (Ed.): CC'99, LNCS 1575, pp. 47–63, 1999.

edges are labeled with *functions* instead of *values*. In effect, the use of edge-functions allows edges to be numbered differently depending on the calling context.

At runtime, as each edge e is traversed, the profiling machinery uses the edge function associated with e to compute a value that is added to the quantity `pathNum`. At the appropriate program points, the profile is updated with the value of `pathNum`.

Because edge functions are always of a particularly simple form (i.e., linear functions), they do not complicate the runtime-instrumentation code greatly:

- The Ball-Larus instrumentation code performs 0 or 1 additions in each basic block; a hash-table lookup and 1 addition for each control-flow-graph backedge; 1 assignment for each procedure call; and a hash-table lookup and 1 addition for each return from a procedure.
- The technique presented in this paper performs 0 or 2 additions in each basic block; a hash-table lookup, 1 multiplication, and 4 additions for each control-flow-graph backedge; 2 multiplications and 2 additions for each procedure call; and 1 multiplication and 1 addition for each return from a procedure.

(The frequency with which our technique and the Ball-Larus technique can avoid performing any additions in a basic block should be about the same.) Thus, while interprocedural path profiling will involve more overhead than intraprocedural path profiling via the Ball-Larus technique, the overheads should not be prohibitive.

The specific technical contributions of this paper include:

- In the Ball-Larus scheme, a cycle-elimination transformation of the (in general, cyclic) control-flow graph is introduced for the purpose of numbering paths. We present the interprocedural analog of this transformation.
- In the case of intraprocedural path profiling, the Ball-Larus scheme produces a dense numbering of the observable paths within a given procedure: That is, in the transformed (i.e., acyclic) version of the control-flow graph for a procedure P, the sum of the edge labels along each path from P's entry vertex to P's exit vertex falls in the range [0..number of paths in P], and each number in the range [0..number of paths in P] corresponds to exactly one such path.

  The techniques presented in this paper produce a dense numbering of interprocedural observable paths. The significance of the dense-numbering property is that it ensures that the numbers manipulated by the instrumentation code have the minimal number of bits possible.

Our work encompasses two main algorithms for interprocedural path profiling, which we call *context path profiling* and *piecewise path profiling*, as well as several hybrid algorithms that blend aspects of the two main algorithms. Context path profiling is best suited for software-maintenance applications, whereas piecewise path profiling is better suited for providing information about interprocedural hot paths, and hence is more appropriate for optimization applications [4].

This paper focuses on context path profiling, and, except where noted, the term "interprocedural path profiling" means "context path profiling". We chose to discuss the context-path-profiling algorithm because the method is simpler to present than the algorithm for piecewise path profiling. However, the same basic machinery is at the heart of both algorithms (see [4]).

The remainder of the paper is organized into four sections: Section 2 presents background material and defines terminology needed to describe our results. Section 3 gives an overview of interprocedural context path profiling. Section 4 describes the technical details of this approach. Section 5 discusses future work.

2 Background

2.1 Supergraph

As in many interprocedural program-analysis problems, we work with an interprocedural control-flow graph called a *supergraph*. Specifically, a program's supergraph G^* consists of a unique entry vertex $Entry_{global}$, a unique exit vertex $Exit_{global}$, and a collection of control-flow graphs (one for each procedure), one of which represents the program's main procedure. For each procedure P, the flowgraph for P has a unique entry vertex, $Entry_P$, and a unique exit vertex, $Exit_P$. The other vertices of the flowgraph represent statements and predicates of the program in the usual way,[1] except that each procedure call in the program is represented in G^* by two vertices, a *call* vertex and a *return-site* vertex. In addition to the ordinary intraprocedural edges that connect the vertices of the individual control-flow graphs, for each procedure call (represented, say, by call vertex c and return-site vertex r) to procedure P, G^* contains a *call-edge*, $c \rightarrow Entry_P$, and a *return-edge*, $Exit_P \rightarrow r$. The supergraph also contains the edges $Entry_{global} \rightarrow Entry_{main}$ and $Exit_{main} \rightarrow Exit_{global}$. An example of a supergraph is shown in Fig. 1(a).

For purposes of profiling, we assume that all branches are logically independent, *i.e.*, the result of one branch does not affect the ability to take any other branch. However, we do not wish to consider paths in G^* that violate the nature of procedure calls (as the path in Fig. 1(b) does). We now develop a language for describing the set of paths in G^* that we wish to consider valid. To do this, let each call site be assigned a unique index between 1 and *NumCallSites*, where *NumCallSites* is the total number of call sites in the program. Then, for each call site with index i, let the call-edge from the call site be labeled with the symbol "$(_i$", and let the return-edge to the call site be labeled with the symbol "$)_i$". Let each edge of the form $Entry_{global} \rightarrow Entry_P$ be labeled with the symbol "$(_P$" and each edge of the form $Exit_P \rightarrow Exit_{global}$ be labeled with the symbol "$)_P$". Let all other edges be labeled with the symbol e. Then a path p in G^* is a *same-level valid path* if and only if the string formed by concatenating the labels

[1] The vertices of a flowgraph can represent individual statements and predicates; alternatively, they can represent basic blocks.

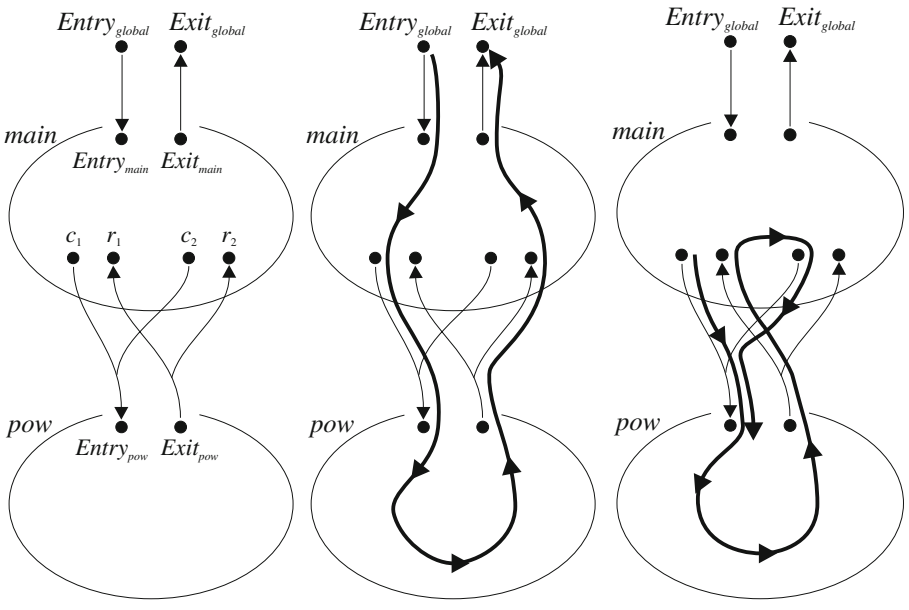

Fig. 1. (a) Schematic of the supergraph of a program in which *main* has two call sites on the procedure *pow*. (b) Example of an invalid path in a supergraph. (c) Example of a cycle that may occur in a valid path.

of p's edges is derived from the non-terminal *SLVP* in the following context-free grammar:

$$\begin{array}{l|l} SLVP ::= e\ \ SLVP & SLVP ::= (_i\ \ SLVP\ \ )_i\ \ SLVP \quad \text{for } 1 \le i \le NumCallSites \\ SLVP ::= \epsilon & SLVP ::= (_P\ \ SLVP\ \ )_P\ \ SLVP \quad \text{for each procedure } P \end{array}$$

Here, ϵ denotes the empty string. A same-level valid path p represents an execution sequence where every call-edge is properly matched with a corresponding return-edge and vice versa.

We also need to describe paths that correspond to incomplete execution sequences in which not all of the procedure calls have been completed. (For example, a path that begins in a procedure P, crosses a call-edge to a procedure Q, and ends in Q.) Such a path p is called an *unbalanced-left path*. The string formed by concatenating the labels on p's edges must be derived from the non-terminal *UnbalLeft* in the following context-free grammar:

$$\begin{array}{l} UnbalLeft ::= UnbalLeft\ \ (_i\ \ UnbalLeft \quad \text{for } 1 \le i \le NumCallSites \\ UnbalLeft ::= UnbalLeft\ \ (_P\ \ UnbalLeft \quad \text{for each procedure } P \\ UnbalLeft ::= SLVP \end{array}$$

2.2 Modifying G^* to Eliminate Backedges and Handle Recursion

For purposes of numbering paths, the Ball-Larus technique modifies a procedure's control-flow graph to remove cycles. This section describes the analogous step for interprocedural context profiling. Specifically, this section describes modifications to G^* that remove cycles from each procedure and from the call graph associated with G^*. The resulting graph is called G^*_{fin}. Each unbalanced-left path through G^*_{fin} defines an "observable path" that can be logged in an interprocedural profile. The number of unbalanced-left paths through G^*_{fin} is finite [4], which is the reason for the subscript "*fin*".

In total, there are three transformations that are performed to create G^*_{fin}. Fig. 3 shows the transformed graph G^*_{fin} that is constructed for the example program in Fig. 2 (the labels on the vertices and edges of this graph are explained in Section 3.1).

Transformation 1: For each procedure P, add a special vertex $GExit_P$. In addition, add an edge $GExit_P \rightarrow Exit_{global}$.

The second transformation removes cycles in each procedure's flow graph. As in the Ball-Larus technique, the procedure's control-flow graph does not need to be reducible; backedges can be determined by a depth-first search of the control-flow graph.

Transformation 2: For each procedure P, perform the following steps:

1. For each backedge target v in P, add a *surrogate* edge $Entry_P \rightarrow v$.
2. For each backedge source w in P, add a *surrogate* edge $w \rightarrow GExit_P$.
3. Remove all of P's backedges.

The third transformation "short-circuits" paths around recursive call sites, effectively removing cycles in the call graph. First, each call site is classified as recursive or nonrecursive. This can be done by identifying backedges in the call graph using depth-first search; the call graph need not be reducible.

Transformation 3: The following modifications are made:

1. For each procedure R called from a recursive call site, add the edges $Entry_{global} \rightarrow Entry_R$ and $Exit_R \rightarrow Exit_{global}$.
2. For each pair of vertices c and r representing a recursive call site that calls procedure R, remove the edges $c \rightarrow Entry_R$ and $Exit_R \rightarrow r$, and add the *summary* edge $c \rightarrow r$. (Note that $c \rightarrow r$ is called a "summary" edge, but not a "surrogate" edge.)

As was mentioned above, the reason we are interested in these transformations is that each observable path—an item we log in an interprocedural path profile—corresponds to an unbalanced-left path through G^*_{fin}. Note that the observable paths should not correspond to just the same-level valid paths through G^*_{fin}: as a result of Transformation 2, an observable path p may end with $\ldots \rightarrow GExit_P \rightarrow Exit_{global}$, leaving unclosed left parentheses. Furthermore, a path in G^*_{fin}that is not unbalanced-left cannot represent any feasible execution path in the original graph G^*.

```
double pow(double base, long exp) {
   double power = 1.0;
   while( exp > 0 ) {
      power *= base;
      exp--;
   }
   return power;
}
```

```
int main() {
   double t, result = 0.0;
   int i = 1;
   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      if( (i%3) == 0 ) {
         t = pow( i, 2 );
         result += t;
      }
      i++;
   }
   return 0;
}
```

Fig. 2. Example program used to illustrate the path-profiling technique. (The program computes the quantity $(\sum_{j=1}^{9}(2 \cdot j)^2)) + (\sum_{k=1}^{6}(3 \cdot k)^2)$.)

Indirect Procedure Calls The easiest way to handle indirect procedure calls is to treat them as recursive procedure calls, and not allow interprocedural paths that cross through an indirect procedure call. Another possibility does allow interprocedural paths to cross through an indirect procedure call: For purposes of numbering the paths in G^*_{fin}, each indirect procedure call through a procedure variable `fp` is turned into an if-then-else chain that has a separate (direct) procedure call for each possible value of `fp`. Well-known techniques (*e.g.*, such as flow insensitive points-to analysis [2,6]) can be used to obtain a reasonable (but still conservative) estimate of the values that `fp` may take on.

3 Overview

In this section, we illustrate, by means of the example shown in Fig. 2, some of the difficulties that arise in collecting an interprocedural path profile. Fig. 1(a) shows a schematic of the supergraph G^* for this program. One difficulty that arises in interprocedural path profiling comes from interprocedural cycles. Even after the transformations described in Section 2.2 are performed (which break intraprocedural cycles and cycles due to recursion), G^* will still contain cyclic paths, namely, those paths that enter a procedure from distinct call sites (see Fig. 1(c)). This complicates any interprocedural extension to the Ball-Larus technique, because the Ball-Larus numbering scheme works on acyclic graphs. There are several possible approaches to overcoming this difficulty:

– One possible approach is to create a unique copy of each procedure for each nonrecursive call site and remove all recursive call and return edges. In our example program, we would create the copies *pow1* and *pow2* of the *pow* function, as shown in Fig. 4. *pow1* can be instrumented as if it had been

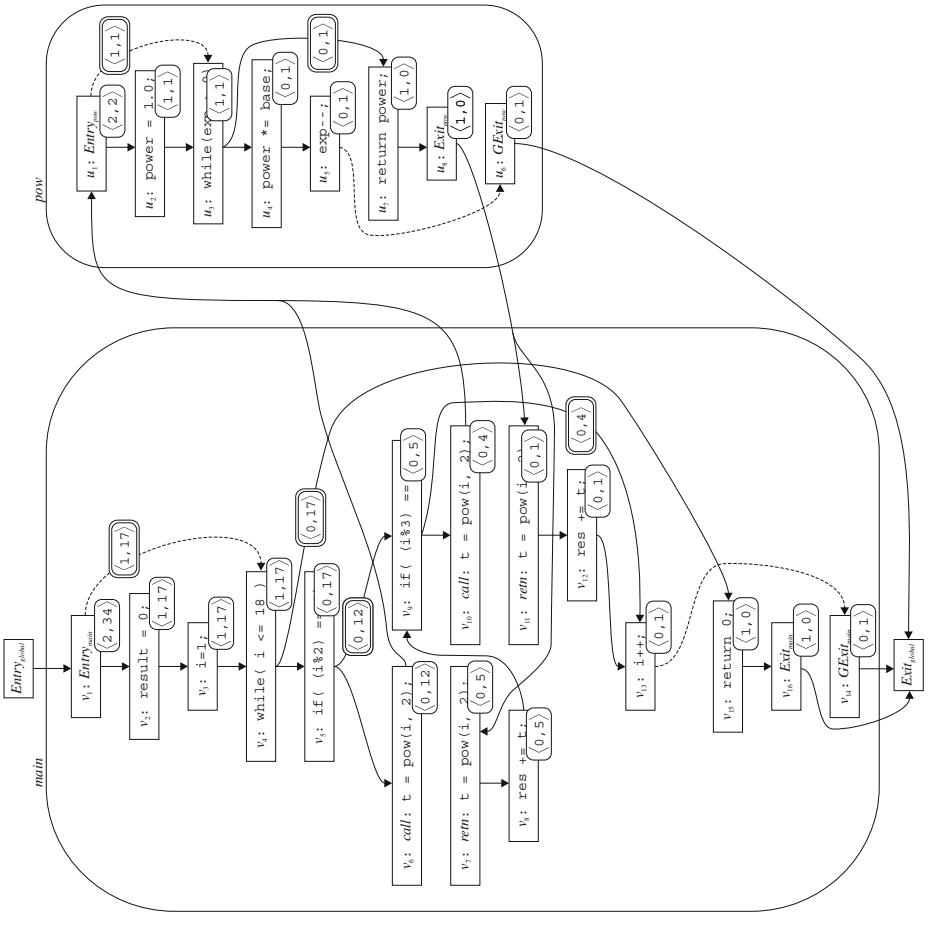

Fig. 3. G^*_{fin} for the code in Fig. 2. Dashed edges represent surrogate edges; the supergraph for the program in Fig. 2 includes the backedges $v_{13} \to v_4$ and $u_5 \to u_3$, which have been removed here by Transformation 2. Here the ordered pair $\langle a, b\rangle$ represents the linear function $\lambda x.a \cdot x + b$. Each vertex v is assigned the linear function ψ_v, which is shown in a rounded box. Each intraprocedural edge e is assigned the linear function ρ_e, which is shown in a doubled, rounded box. Unlabeled intraprocedural edges have the function $\langle 0, 0\rangle$. Interprocedural edges do not have ρ functions.

inlined in *main*, and likewise for *pow2*. In many cases, this approach is impractical because of the resulting code explosion.

– A second approach—which is the one developed in this paper—is to parameterize the instrumentation in each procedure to behave differently for different calling contexts. In our example, *pow* is changed to take an extra parameter. When *pow* is called from the first call site in *main*, the value of

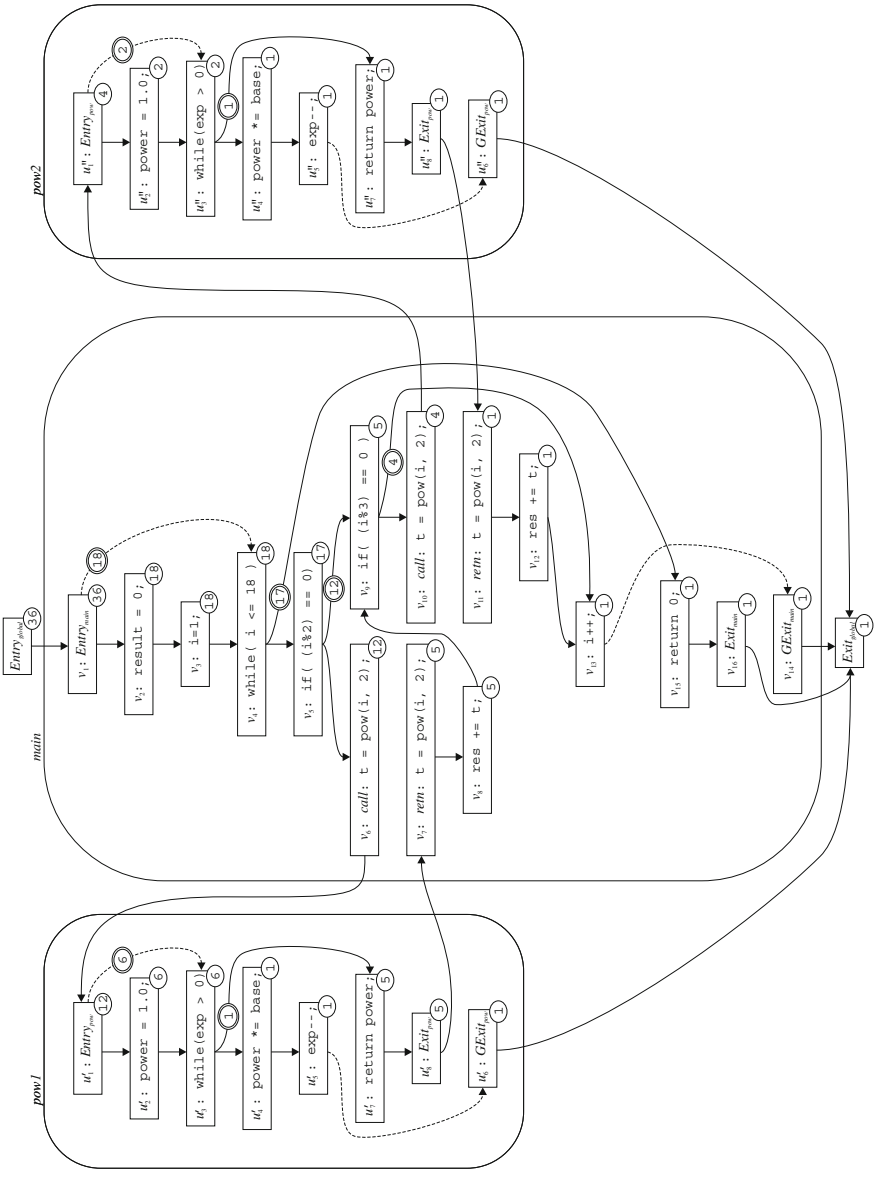

Fig. 4. Modified version of G^*_{fin} from Fig. 3 with two copies of *pow*. Labels on the vertices and edges show the results of applying the Ball-Larus numbering technique to the graph. Each vertex label is shown in a circle, and each edge label are shown in a double circle. Unlabeled edges are given the value 0 by the Ball-Larus numbering scheme.

the new parameter causes the instrumentation of *pow* to mimic the behavior of the instrumentation of *pow1* in the first approach above; when *pow* is called from the second call site in *main*, the value of the new parameter causes *pow*'s instrumentation to mimic the behavior of the instrumentation of *pow2*. Thus, by means of an appropriate parameterization, we gain the advantages of the first approach without duplicating code.

Section 3.1 gives a high-level description of our path-numbering technique and Section 4 gives a detailed description of the profiling algorithm.

3.1 Numbering Unbalanced-Left Paths

Extending the Ball-Larus technique to number unbalanced-left paths in G^*_{fin} is complicated by the following facts:

1. While the number of unbalanced-left paths is finite, an unbalanced-left path may contain cycles (such as those in Fig. 1(c)).
2. The number of paths that may be taken from a vertex v is dependent on the path taken to reach v: for a given path p to vertex v, not every path q from v forms an unbalanced-left path when concatenated with p.

These facts mean that it is not possible to assign a single integer value to each vertex and edge of G^*_{fin} as the Ball-Larus technique does. Instead, each occurrence of an edge e in a path p will contribute a value to the path number of p, but the value that an occurrence of e contributes will be dependent on the part of p that precedes that occurrence of e. In particular, e's contribution is determined by the sequence of unmatched left parentheses that precede the occurrence of e in p. (The sequence of unmatched left parentheses represents a calling context of the procedure containing e.)

Consider the example shown in Figs. 2 and 3. Notice that G^*_{fin} in Fig. 3 contains cyclic, unbalanced-left paths. For example, the following path is a cycle from u_1 to u_1 that may appear as a subpath of an unbalanced-left path:

$$u_1 \rightarrow u_3 \rightarrow u_7 \rightarrow u_8 \rightarrow v_7 \rightarrow v_8 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1.$$

Fig. 4 shows a modified version of G^*_{fin} with two copies of the procedure *pow*, one for each call site to *pow* in *main*. This modified graph is acyclic and therefore amenable to the Ball-Larus numbering scheme: Each vertex v in Fig. 4 is labeled with $numPaths[v]$, the number of paths from v to $Exit_{global}$; each edge e is labeled with its Ball-Larus increment [3]. Note that there is a one-to-one and onto mapping between the paths through the graph in Fig. 4 and the unbalanced-left paths through the graph in Fig. 3. This correspondence can be used to number the unbalanced-left paths in Fig. 3: each unbalanced-left path p in Fig. 3 is assigned the path number of the corresponding path q in Fig. 4.

The following two observations capture the essence of our technique:

- Because the labeling passes of the Ball-Larus scheme work in reverse topological order, the values assigned to the vertices and edges of a procedure are dependent upon the values assigned to the exit vertices of the procedure. For instance, in Fig. 4, the values assigned to the vertices and edges of *pow1* are determined by the values assigned to $Exit_{pow1}$ and $GExit_{pow1}$ (*i.e.*, the values 5 and 1, respectively), while the values assigned to the vertices and edges of *pow2* are determined by the values assigned to $Exit_{pow2}$ and $GExit_{pow2}$ (*i.e.*, the values 1 and 1, respectively). Note that $numPaths[GExit_P] = 1$ for any procedure P (since the only path from $GExit_P$ to $Exit_{global}$ is the path consisting of the edge $GExit_P \rightarrow Exit_{global}$). Thus, the values on the edges and the vertices of *pow1* differ from some of the values on the corresponding edges and vertices of *pow2* because $numPaths[Exit_{pow1}] \neq numPaths[Exit_{pow2}]$.
- Given that a program transformation based on duplicating procedures is undesirable, a mechanism is needed that assigns vertices and edges different numbers depending on the calling context. To accomplish this, each vertex u of each procedure P is assigned a linear function ψ_u that, when given a value for $numPaths[Exit_P]$, returns the value of $numPaths[u]$. Similarly, each edge e of each procedure P is assigned a linear function ρ_e that, when given a value for $numPaths[Exit_P]$, returns the Ball-Larus value for e.

Fig. 3 shows G^*_{fin} labeled with the appropriate ψ and ρ functions. Note that we have the desired correspondence between the linear functions in Fig. 3 and the integer values in Fig. 4. For example, in Fig. 3 vertex u_1 has the function $\psi_{u_1} = \lambda x.2 \cdot x + 2$. This function, when supplied with the value $numPaths[Exit_{pow1}] = 5$ from Fig. 4 evaluates to 12, which is equal to $numPaths[u'_1]$ in Fig. 4. However, when $\lambda x.2 \cdot x + 2$ is given the value $numPaths[Exit_{pow2}] = 1$, it evaluates to 4, which is equal to $numPaths[u''_1]$ in Fig. 4.

To collect the number associated with an unbalanced-left path p in G^*_{fin}, as p is traversed, each edge e contributes a value to p's path number. As illustrated below, the value that e contributes is dependent on the path taken to e:

Example 1. Consider the edge $u_1 \rightarrow u_3$ in G^*_{fin}, and an unbalanced-left path s that begins with the following path prefix:

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \tag{1}$$

In this case, the edge $u_1 \rightarrow u_3$ contributes a value of 6 to s's path number. To see that this is the correct value, consider the path prefix in Fig. 4 that corresponds to (1):

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u'_1 \rightarrow u'_3$$

In Fig. 4, the value on the edge $u'_1 \rightarrow u'_3$ is 6.

In contrast, in an unbalanced-left path t that begins with the path prefix

$$Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_9 \rightarrow v_{10} \rightarrow u_1 \rightarrow u_3 \tag{2}$$

the edge $u_1 \rightarrow u_3$ will contribute a value of 2 to t's path number. (To see that this is the correct value, consider the path prefix in Fig. 4 that corresponds to (2).)

It can even be the case that an edge e occurs more than once in a path p, with each occurrence contributing a different value to p's path number. For example, there are some unbalanced-left paths in G^*_{fin} in which the edge $u_1 \rightarrow u_3$ appears twice, contributing a value of 6 for the first occurrence and a value of 2 for the second occurrence.

To determine the value that an occurrence of the edge e should contribute to a path number, the profiling instrumentation will use the function ρ_e and the appropriate value for $numPaths[Exit_P]$, where P is the procedure containing e. Thus, as noted above, an occurrence of the edge $u_1 \rightarrow u_3$ may contribute the value $(\lambda x.x + 1)(1) = 2$ or the value $(\lambda x.x + 1)(5) = 6$ to a path number, depending on the path prior to the occurrence of $u_1 \rightarrow u_3$.

```
unsigned int profile[36];    /* 36 possible paths in total */
double pow(double base, long exp,
           unsigned int &pathNum, unsigned int numValidCompsFromExit){
   unsigned int pathNumOnEntry = pathNum; /* Save the calling context */
   double power = 1.0;
   while( exp > 0 ) {
      power *= base;
      exp--;
      profile[pathNum]++;
      /* From surrogate edge u1->u3: */
      pathNum = 1 * numValidCompsFromExit + 1 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 1; /* From edge u3->u7 */
   return power;
}
```

Fig. 5. Part of the instrumented version of the program from Fig. 2. Instrumentation code is shown in italics. (See also Fig. 6.)

Figs. 5 and 6 show the program from Fig. 2 with additional instrumentation code — based on the linear functions in Fig. 3 — that collects an interprocedural path profile. The output from the instrumented program is as follows:

```
 0: 0    1: 0    2: 0    3: 0    4: 0    5: 0    6: 0    7: 0    8: 0
 9: 0   10: 0   11: 0   12: 0   13: 0   14: 0   15: 0   16: 1   17: 0
18: 9   19: 0   20: 0   21: 0   22: 0   23: 0   24: 9   25: 3   26: 0
27: 3   28: 3   29: 6   30: 3   31: 0   32: 3   33: 3   34: 5   35: 1
```

Section 4 presents an algorithm that assigns linear functions to the vertices and edges of G^*_{fin} directly, without referring to a modified version of G^*_{fin}, like the one shown in Fig. 4, in which procedures are duplicated.

```
int main() {
   unsigned int pathNum = 0;
   unsigned int pathNumOnEntry = 0;
   unsigned int numValidCompsFromExit = 1;
   double t, result = 0.0;
   int i = 1;
   while( i <= 18 ) {
      if( (i%2) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 5 );
         /* On entry to pow: pathNum is 0 or 18; fourth arg. always 5 */
         /* On exit from pow: pathNum is 1, 7, 19, or 25 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 12;
      if( (i%3) == 0 ) {
         t = pow( i, 2, pathNum, 0 * numValidCompsFromExit + 1 );
         /* On entry to pow: pathNum is 1, 7, 12, 19, 25, or 30; 4th arg. always 1 */
         /* On exit from pow: pathNum is 2, 3, 8, 9, 13, 14, 20, 21, 26, 27, 31, or 32 */
         result += t;
      } else
         pathNum += 0 * numValidCompsFromExit + 4;     /* From edge v9->v13 */
      i++;
      profile[pathNum]++;
      /* From surrogate edge v1->v4: */
      pathNum = 1 * numValidCompsFromExit + 17 + pathNumOnEntry;
   }
   pathNum += 0 * numValidCompsFromExit + 17;          /* From edge v4->v15 */
   profile[pathNum]++;
   for (i = 0; i < 36; i++) {
      cout.width(3); cout << i << ":"; cout.width(2); cout << profile[i] << "  ";
      if ((i+1) % 9 == 0) cout << endl;
   }
   return 0;
}
```

Fig. 6. Part of the instrumented version of the program from Fig. 2. Instrumentation code is shown in italics. (See also Fig. 5.)

3.2 What Do You Learn From a Profile of Unbalanced-Left Paths?

Before examining the details of interprocedural path profiling, it is useful to understand the information that is gathered in this approach:

- Each unbalanced-left path p through G^*_{fin} from $Entry_{global}$ to $Exit_{global}$ can be thought of as consisting of a *context-prefix* and an *active-suffix*. The active-suffix q'' of p is a maximal-size, surrogate-free subpath at the tail of p (though the active-suffix may contain summary edges of the form $c \rightarrow r$, where c and r represent a recursive call site). The context-prefix q' of p is the prefix of p that ends at the last surrogate edge before p's active suffix. (The context-prefix q' can be the empty path from $Entry_{global}$ to $Entry_{global}$.)
- The counter associated with the unbalanced-left path p counts the number of times during a program's execution that the active-suffix of p occurs in the context summarized by p's context-prefix.

Example 2. Consider the path in Fig. 3 with path number 24:

$$24: Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5 \rightarrow u_6 \rightarrow Exit_{global}$$

This path consists of the context-prefix $Entry_{global} \rightarrow v_1 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow u_1$ and the active-suffix $u_3 \rightarrow u_4 \rightarrow u_5$. The output obtained from running the program shown in Figs. 5 and 6 indicates that the active suffix was executed 9 times in the context summarized by the context-prefix. Note that the context-prefix not only summarizes the call site in *main* from which *pow* was called, but also the path within *main* that led to that call site. In general, a context-prefix (in an interprocedural technique) summarizes not only a sequence of procedure calls (*i.e.*, the calling context), but also the intraprocedural paths taken within each procedure in the sequence.

4 Interprocedural Path Profiling

In this section, we discuss the ψ and ρ functions that serve as replacements for the vertex and edge values of the Ball-Larus technique.

4.1 Assigning ψ and ρ Functions

Solving for ψ Functions For a vertex v in procedure P, the function ψ_v takes the number of valid completions from $Exit_P$ (for an unbalanced-left path p to $Entry_P$ concatenated with any same-level valid path to $Exit_P$) and returns the number of valid completions from v (for the path p concatenated with any same-level valid path to v).

We can find the ψ functions by setting up and solving a collection of equations. For an exit vertex $Exit_P$, ψ_{Exit_P} is the identity function: $\psi_{Exit_P} = id$. For a vertex of the form $GExit_P$, we have the equation $\psi_{GExit_P} = \lambda x.1$. This equation reflects the fact that the number of valid completions from $GExit_P$ is always 1, regardless of the number of valid completions from $Exit_P$. For a call vertex c to a procedure Q associated with the return-site vertex r, where c and r represent a non-recursive call site, we have the equation $\psi_c = \psi_{Entry_Q} \circ \psi_r$. For all other cases, for a vertex m, we have the equation $\psi_m = \sum_{n \in succ(m)} \psi_n$, where $succ(m)$ denotes the successors of m, and the addition $f + g$ of function values f and g is defined to be the function $\lambda x.f(x) + g(x)$.[2]

Because $id(= \lambda x.x)$ and $\lambda x.1$ are both linear functions of one variable, and the space of linear functions of one variable is closed under function composition and function addition, each ψ function is a linear function of one variable. Furthermore, each ψ function $\lambda x.a \cdot x + b$ can be represented as an ordered pair $\langle a, b \rangle$.

To find ψ functions that satisfy the above equations, each procedure P is visited in reverse topological order of the call graph, and each vertex v in P is

[2] The equations for the ψ functions closely resemble the ϕ functions of Sharir and Pnueli's functional approach to interprocedural data-flow analysis [4,5].

visited in reverse topological order of P's control-flow graph. (For purposes of ordering the vertices of a procedure P, a return-site vertex r is considered to be a successor of its associated call vertex c.) As each vertex v is visited, the appropriate equation given above is used to determine the function ψ_v.

The order of traversal guarantees that when vertex v is visited, all of the functions that are needed to determine ψ_v will be available. This follows from the fact that the call graph associated with G^*_{fin} is acyclic and the fact that the flow graph of each procedure in G^*_{fin} is acyclic. (The fact that the call graph and flow graphs are acyclic also explains why each vertex needs to be visited only once.)

Solving for ρ functions Each intraprocedural edge e in procedure P is assigned a linear function ρ_e. The function ρ_e, when supplied with the number of valid completions from $Exit_P$ (for an unbalanced-left path p to $Entry_P$ concatenated with any same-level valid path from $Entry_P$ to $Exit_P$), returns the value that e contributes (to the path number of the path p concatenated with any same-level valid path to e).

Let v be an intraprocedural vertex that is the source of one or more intraprocedural edges. (Note that v cannot be a call vertex for a nonrecursive call site, nor have the form $Exit_P$, nor have the form $GExit_P$.) Let $w_1 \ldots w_k$ be the successors of v. Then we make the following definition:

$$\rho_{v \to w_i} = \begin{cases} \lambda x.0 & \text{if } i = 1 \\ \sum_{j<i} \psi_{w_j} & \text{otherwise} \end{cases} \tag{3}$$

Clearly, each ρ function is a linear function of one variable. Furthermore, (3) can be used to find each ρ function when the ψ functions are known.

4.2 Computing Values for Interprocedural Edges

Unlike intraprocedural edges, an interprocedural edge e always contributes the same value, independent of the path taken to e [4]. For interprocedural edges that are not of the form $Entry_{global} \to Entry_P$, this value is always 0.

For each edge $Entry_{global} \to Entry_P$ and each unbalanced-left path p that starts with this edge, we define the integer value $edgeValue[Entry_{global} \to Entry_P]$ to be the value that $Entry_{global} \to Entry_P$ contributes to p's path number. To find the *edgeValue* values, it is necessary to use a fixed (but arbitrary) ordering of the edges of the form $Entry_{global} \to Entry_P$. For convenience, we number each edge $Entry_{global} \to Entry_P$ according to this ordering, and use the notation Q_i to refer to the procedure that is the target of the i^{th} edge. We have the following:

$$edgeValue[Entry_{global} \to Entry_{Q_i}] = \begin{cases} 0 & \text{if } i = 0 \\ \sum_{j<i} \psi_{Entry_{Q_j}}(1) & \text{otherwise} \end{cases}$$

4.3 Calculating the Path Number of an Unbalanced-Left Path

In this section, we show how to calculate the path number of an unbalanced-left path p through G^*_{fin} from $Entry_{global}$ to $Exit_{global}$. This is be done during

a single traversal of p that sums the values contributed by each edge e for each path prefix p' such that $[p' \parallel e]$ is a prefix of p.

For an interprocedural edge e, the value $edgeValue[e]$ contributed by e is calculated as described in Section 4.2. For an intraprocedural edge e in procedure P, the value contributed by e (for the path p' leading to e) is calculated by applying the function ρ_e to the number of valid completions from $Exit_P$. (The number of valid completions from $Exit_P$ is determined by the path taken to $Entry_P$—in this case a prefix of p'.)

We now come to the crux of the matter: how to determine the contribution of an edge e when the edge is traversed without incurring a cost for inspecting the path p' taken to e. The trick is that, as p is traversed, we maintain a variable, `numValidCompsFromExit`, that holds the number of valid completions from the exit vertex $Exit_Q$ of the procedure Q that is currently being visited. The number of valid completions from $Exit_Q$ is uniquely determined by p'—specifically, the sequence of unmatched left parentheses in p'. The value `numValidCompsFromExit` is maintained by the use of a stack, `NVCStack`, and the ψ functions for return-site vertices. The following steps describe the algorithm to compute the path number for a path p (this number is accumulated in the variable `pathNum`):

- When the traversal of p is begun, `numValidCompsFromExit` is set to 1. This indicates that there is only one valid completion from $Exit_R$, where R is the first procedure that p enters: if p reaches the exit of the first procedure it enters, then it must follow the edge $Exit_P \rightarrow Exit_{global}$. The value of `pathNum` is initialized to the value $edgeValue[e]$ on the first edge e of p (see Section 4.2).
- As the traversal of p crosses a call-edge $c \rightarrow Entry_T$ from a procedure S to a procedure T, the value of `numValidCompsFromExit` is pushed on the stack, and is updated to ψ_r(`numValidCompsFromExit`), where r is the return-vertex in S that corresponds to the call-vertex c. This reflects the fact that the number of valid completions from $Exit_T$ is equal to the number of valid completions from r.
- As the traversal of p crosses a return-edge $Exit_T \rightarrow r$ from a procedure T to a procedure S, the value of `numValidCompsFromExit` is popped from the top of the stack. This reflects the fact that the number of valid completions from the exit of the calling procedure S is unaffected by the same-level valid path through the called procedure T.
- As the traversal of p crosses an intraprocedural edge e, the value of `pathNum` is incremented by ρ_e(`numValidCompsFromExit`).
- At the end of the traversal of p, `pathNum` holds the path number of p.

4.4 Runtime Environment for Collecting a Profile

We are now ready to describe the instrumentation code that is introduced to collect an interprocedural path profile. In essence, the instrumentation code threads the algorithm described in Section 4.3 into the code of the instrumented program. Thus, the variables `pathNum` and `numValidCompsFromExit` become program variables. There is no explicit stack variable corresponding to `NVCstack`;

instead, `numValidCompsFromExit` is passed as a value-parameter to each procedure and the program's execution stack is used in place of `NVCstack`. The instrumentation also makes use of two local variables in each procedure:

`pathNumOnEntry` stores the value of `pathNum` on entry to a procedure. When an intraprocedural backedge is traversed in a procedure P, the instrumentation code increments the count associated with the current observable path and begins recording a new observable path that has the context-prefix indicated by the value of `pathNumOnEntry`.

`pathNumBeforeCall` stores the value of `pathNum` before a recursive procedure call is made. When the recursive procedure call is made, the instrumentation begins recording a new observable path. When the recursive call returns, the instrumentation uses the value in `pathNumBeforeCall` to resume recording the observable path that was executing before the call was made.

Figs. 5 and 6 show an instrumented version of the code in Fig. 2. Reference [4] gives a detailed description of the instrumentation used to collect an interprocedural path profile and describes how the intrumentation can be made more efficient than the code shown in Figs. 5 and 6.

5 Future Work

We are currently in the process of implementing the algorithm described in the paper, and thus do not yet have performance figures to report. The main reasons for believing that the technique described (or a variation on it) will prove to be practical are:

- The Ball-Larus technique for intraprocedural profiling has very low overhead (31% on the SPEC benchmarks [3]). As discussed in the Introduction, although interprocedural path profiling involves more overhead than the Ball-Larus technique, the additional overhead should not be prohibitive.
- In the worst case, the number of paths through a program is exponential in the number of branch statements b, and thus the number of bits required to represent paths is linear in b. However, as in the Ball-Larus approach, it is possible to control the explosion in the number of paths by altering G^*_{fin} to remove paths from it (and adjusting the instrumentation code accordingly). There are a variety of techniques that can be applied without having to fall back on pure intraprocedural profiling [4].

Acknowledgements

This work was supported in part by the NSF under grants CCR-9625667 and CCR-9619219, by an IBM Partnership Award, by a Vilas Associate Award from the Univ. of Wisconsin, and by the "Cisco Systems Wisconsin Distinguished Graduate Fellowship".

References

1. G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI'97*, June 1997. 47
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, Univ. of Copenhagen, May 1994. (DIKU report 94/19). 52
3. T. Ball and J. Larus. Efficient path profiling. In *MICRO 1996*, 1996. 47, 55, 62
4. D. Melski and T. Reps. Interprocedural path profiling. Tech. Rep. TR-1382, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, September 1998. Available at "http://www.cs.wisc.edu/wpis/papers/tr1382.ps". 48, 49, 51, 59, 60, 62
5. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981. 59
6. B. Steensgaard. Points-to analysis in almost-linear time. In *Symp. on Princ. of Prog. Lang.*, pages 32–41, 1996. 52