Know your place:*
Selectively executing statements based on context

Ben Wiedermann
(ben@cs.utexas.edu)

May 14, 2004

1 Introduction

This project provides a way for programmers to insert runtime context-sensitive code into their C programs. The idea is that a program will execute a piece of code only if the program is in a given calling context (i.e., a stack of procedure calls). One way to accomplish this technique is for the program to examine the runtime stack, determine if it matches a given context, then execute the context-sensitive code. Of course, this method is extremely inefficient.

A more efficient way to determine the context is to staticly compute a unique "context value" for every possible context along a chain of procedure calls. At runtime, the program compares the actual context number to the desired context number. If these numbers are equivalent, the program executes the context-sensitive code.

This facility can be useful in several scenarios:

Checkpointing A programmer may want to insert checkpoints at certain locations in the program, but *only* if the program is in a certain context.

Profiling A programmer could use this technique to implement efficient, context-sensitive profiling. The program would simply write out the context number at appropriate points.

Context-sensitive garbage collection Garbage often occurs in phases: a program allocates an amount of memory, uses it for awhile, then the memory becomes garbage. Very often, the behavior coincides with a certain stack behavior (i.e., a procedure exits and a large chunk of memory becomes available). A profiling technique could be combined with this context-sensitive technique to explicitly trigger a collection at given points.

This project describes a modification to the Ball-Larus path profiling technique [1] that provides a way to compute the context value, as described above. The technique is

*Completed as a project for Dr. Calvin Lin's CS380c course at The Univeristy of Texas at Austin.

implemented as a series of C-Breeze walkers and changers that modify C code to keep track of a running context value. The programmer can then insert selective statements in the modified code.

The remainder of this report briefly describes the Ball Larus technique (Section 2) before detailing the changes necessary to track contexts (Section 3). Section 4 describes the C-Breeze implementation, and Section 5 gives an example of how to insert context-sensitive code. We conclude with a section on future work (Section 6), which describes some ways in which the technique can be improved.

2 The Ball-Larus Path Profiling Technique

The Ball-Larus path profiling technique efficiently records how frequently a program executes a control-flow path. Their technique annotates the edges of the program's control-flow graph with integer "modifiers", that — when summed over a path — produce a unique value for that path. In this way, the technique assigns a unique integer value to each path through the program. To compute the frequency of each path's execution, Ball and Larus then describe a way to instrument the program code such that the program increments a path-specific counter every time the program executes the corresponding path. The algorithm that computes control-flow graph modifiers is as follows:

---

for all node n in reverse topological order **do**
  if n is a leaf **then**
    NumPaths(n) $\leftarrow$ 1
  else
    NumPaths(n) $\leftarrow$ 0
    for all edge e of the form $n \rightarrow m$ **do**
      Val(e) $\leftarrow$ NumPaths(n)
      NumPaths(n) $\leftarrow$ NumPaths(n) + NumPaths(m)
    end for
  end if
end for

Figure 1 shows a simple control-flow graph, in which there are four paths. Applying the algorithm above yields the edge labels in the graph. Note that, at each leaf, the sum of the modifiers along the edges to that leaf yield a unique path number (zero through three). The Ball-Larus technique instruments the leaves to record which path has been taken.

The above algorithm works only on directed *non-cyclic* graphs. Ball and Larus handle loops by effectively breaking a program's paths on loop boundaries. The technique tracks separate paths from before the loop through the loop body, from the loop entry through the loop and from the loop exit through the remainder of the program. The point here is not the details of their technique's handling of loops, but rather the

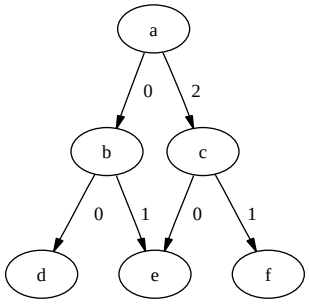

Figure 1: A simple control-flow graph

idea that cycles in the graph pose a problem that we must handle when we adapt the technique to callsite graphs.

3 Deriving Contexts from a Callsite Graph

Our approach adapts the Ball-Larus path profiling algorithm to enumerate a program's possible contexts. As a continuing example, consider the program in Fig. 2, whose callsite graph also appears in the figure.

```
/* example.c */

void f() {}

void e() {}

void d() {}

void c()
{e(); f();}

void b()
{d(); e();}

void a()
{b(); c();}

void main()
{a();}
```

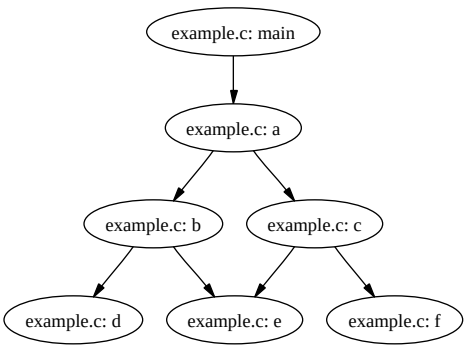

Figure 2: Simple program and its callsite graph

Our goal is to assign a unique modifier to each edge in the graph, so that, at any given node, we can determine the path (context) by which that node has been reached. Certainly, we could directly adapt the Ball-Larus algorithm from the previous section to operate over callsite graphs, instead of control-flow graphs. If we were to do so, then we would obtain identical modifiers for the graph in Fig. 2 as for the graph in Fig. 1. However, there is a key difference between identifying paths and identifying contexts. When identifying paths, it is a branch that creates the possibility of a multiple paths. When identifying contexts, it is a merge that creates the possibility of multiple contexts.

Therefore, we do not need to instrument the edge `b` $\rightarrow$ `d` (as we did in Fig. 1), because there is only one possible context for function `d` (namely `main` $\rightarrow$ `a` $\rightarrow$ `b` $\rightarrow$ `d`). Similarly, we do not need to instrument either edge `a` $\rightarrow$ `b` or edge `a` $\rightarrow$ `c` (as we did in Fig. 1), because these two edges do not imply different contexts for their targets. We *do*, however, need to instrument either edge `b` $\rightarrow$ `e` and edge `c` $\rightarrow$ `e`, because there are two possible contexts in which `e` can be called.

Another key difference is that, while control-flow graphs are directed graphs, callsite graphs are directed *multi*graphs, because one function may call another function multiple times. We may want to differentiate among multiple callsites, and must be able to capture this information in our analysis.

We can formalize the differences between the two techniques by defining the following algorithm:

---

for all node n in topological order **do**
  if n is a leaf **then**
    NumContexts(n) $\leftarrow$ 1
  else
    currentContext $\leftarrow$ NumContexts(n)
    for all edge e of the form $m \rightarrow n$ **do**
      Val(e) $\leftarrow$ currentContext
      currentContext $\leftarrow$ currentContext + NumContexts(m)
    end for
    NumContexts(n) $\leftarrow$ currentContext
  end if
end for

Performing this algorithm over the graph in Fig. 2 gives the annotated graph shown in Fig. 3. One benefit of this algorithm is that we represent the context number minimally, because we can reuse context numbers for independent nodes (e.g., functions `d` and `f`).

Cycles (i.e., recursion) in a callsite graph create problems, because we cannot topologically sort a cyclic, directed graph. However, recursion is a common programming technique, so we must find some way for our analysis to handle it. A sensible solution, for a simple recusive function, might be to record whether the program recursed before reaching some node. However, this solution proved too complex to implement,

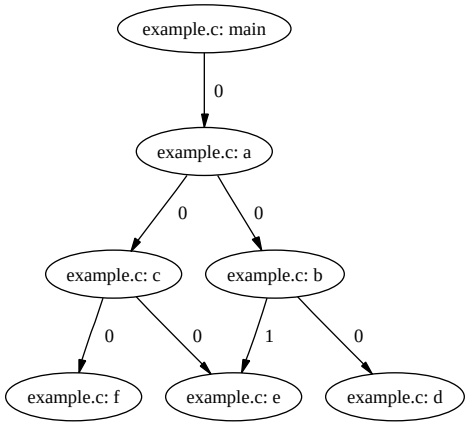

Figure 3: Annotated version of graph in Fig. 2

given our simple, modified Ball-Larus algorithm. Instead, we choose to collapse the strongly-connected compontents of the callsite graph and perform the alogorithm over the resulting, acyclic graph.

Figure 4 contains a more complex version of the program in Fig 2, along side its callsite graph and its modified, annotated control-flow graph. Notice that the algorithm handles recursion as well as multiple calls to the same function from within another function. The key observation about our approach to recursion is that we retain all information about how we reached a strongly connected component and how we exited a strongly connected component, but we lose all information about what occurs inside the strongly connected component. In other words, recursion becomes a black box in terms of contexts. For a more detailed discussion of the recursion issue, with suggestions for improvement, see Section 6.

4 Implementation

This project is implemented as a series of C-Breeze [3] `Walker`s and `Changer`s. We added two phases: `context` and `csg`, which changes the compiled code to keep track of the context number and which prints notation for DOT [2] graphs, respectively. We use a custom graph implementation, specialized for the callsite graph and operations the phases perform on it. In this section, we overview the specialized implementation of the callsite graph, the C-Breeze `Walker` for annotating the graph with context values, the `Changer` for keeping track of the context number in the code, and the `Walker` for printing the DOT notation for the annotated graph. We conclude the section with a discussion of the performance characteristics of the various program elements.

```
/* example2.c */

void a(); void b(int); void c(int);
void d(); void e(); void f();

void f() {}

void e() {}

void d() {}

void c(int i)
{if (i!=0) b(0);
 e(); f();}

void b(int i)
{c(i); d();
 e(); d();}

void a()
{b(1); c(1);}

void main()
{a();}
```

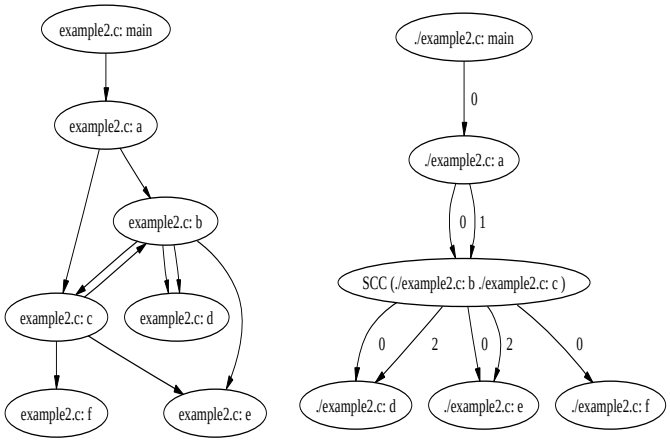

Figure 4: Program with recursion, callsite graph, and annotated modified callsite graph

4.1 Representing Graphs

We use two classes to represent graphs: `CallSiteGraph` and `ContextGraph`. Class `CallSiteGraph` represents a basic graph, where the nodes correspond to functions and an edge corresponds to a call from one function to another. Class `ContextGraph` subclasses `CallSiteGraph`, to provide functionality for a `CallSiteGraph` annotated according to the modified Ball-Larus algorithm.

4.1.1 `CallSiteGraph`

We represent the callsite graph abstractly as a pair `(Nodes, Edges)`, where `Nodes` is a list of `procNode`s and an edge exists between nodes `A` and `B` if function `A` calls function `B`. Thus, the graph is a directed, possibly cyclic multigraph. Every node has a `name` that is a string. Every edge has a label that is an `integer` and corresponds to the modifier for that edge. Table 1 lists the available `CallSiteGraph` operations.

Name	Return Value	Description
addNode(node)	void	adds a new node to the graph
getNodes()	list of nodes	returns a list of all nodes in the graph
name(node)	string	given a node, returns its name
name(node, string)	void	given a node and a new name, renames the node
addEdge(node, node, callNode *)	void	adds edge from one node to another, with the corresponding callNode
deleteEdge(edge)	void	removes a given edge from the graph
getEdges()	list of edges	returns a list of all edges in the graph
getPreds(node)	list of edges	given a node, returns a list of the node's incoming edges
getSuccs(node)	list of edges	given a node, returns a list of the node's outgoing edges
label(edge)	int	given an edge, returns the edge's label
label(edge, int)	void	given an edge and a new label, changes the edge's label
getSource(edge)	node	returns the source of the edge
getTarget(edge)	node	returns the target of the edge
getCallNode(edge)	callNode *	returns a pointer to the callNode that corresponds to a given edge
topsort()	list of nodes	if the graph is acyclic, returns a list of nodes in topological order
findSCC()	list of lists of nodes	returns lists that contain nodes for each strongly-connected component
collapseSCC()	void	collapses the strongly connected components of the graph
printDot()	void	prints to standard output the DOT representation of the graph

Table 1: CallSiteGraph operations

4.1.2 ContextGraph

Class ContextGraph provides an interface to a CallSiteGraph annotated with context values. The algorithm method of this class annotates the callsite graph, according to the algorithm described in Section 3. Table 2 lists the available ContextGraph operations.

Name	Return Value	Description
context(node)	int	returns the number of contexts through which a given node can be reached
context(node, int)	void	sets the number of contexts through which a given node can be reached
modifier(callNode *)	int	returns the annotation for the edge that corresponds to the given callNode
modifier(callNode *, int)	void	sets the annotation for the edge that corresponds to the given callNode
algorithm()	void	executes the algorithm from Section 3 on the ContextGraph

Table 2: ContextGraph operations

4.2 Building the Callsite Graph

Class ContextWalker is a C-Breeze walker that builds the callsite graph and runs the algorithm described in 3. The walker provides one static method buildContextGraph that performs the walker's work, and returns an object of class ContextGraph, which represents the annotated callsite graph. To perform its work, method buildContextGraph executes the following steps:

initialize list of functions F to be $\emptyset$
initialize a `ContextGraph` cw to be $(\emptyset, \emptyset)$
for all $u \in$ units **do**
  for all $d \in u$'s defintitions **do**
    if d is a function definition **then**
      add d to F
      add d to the nodes of cw
    end if
  end for
end for
for all $f \in F$ **do**
  for all $f' \in F$ such that f calls f' **do**
    add $f \rightarrow f'$ to edges of cw, with the corresponding `callNode`
  end for
end for
run Section 3's algorithm on cw

C-Breeze phases can use the `ContextWalker` by calling `buildContextGraph`, then performing work over the resulting `ContextGraph` object. In this report, we discuss two such phases: `context` (Section 4.3) and `csg` (Section 4.4).

4.3 Keeping Track of the Context Number

This project provides the `context` phase that, given a C program, will insert code in functions that updates a global context number, according to an annotated `ContextGraph`. To execute this code on a file, use the following command:

```
cbz -context -c-code <file>
```

Class `ContextChanger` is a C-Breeze changer that peforms the work of the `context` phase. This phase is simple, and performs the following steps:

perform Section 4.2's algorithm, to obtain `ContextGraph` cw
for all $u \in$ units **do**
  add definition `int __context := 0` to the top of u
  for all $c \in$ callsites **do**
    $modifier \leftarrow c$'s modifier, according to cw
    if $modifier > 0$ **then**
      change c's statement `call(...)` to
        `__context +=` $modifier$
        `call(...)`
        `__context -=` $modifier$
    end if
  end for
end for

Figure 5 shows the results of running the `context` phase on the program from Fig. 2. Notice that we change a call only when its modifier is greater than 0, making the resulting code slightly more efficient.

```
int __context = 0;

void f(void) {}

void e(void) {}

void d(void) {}

void c(void)
{e(); f();}

void b(void)
{
  d();
  {
    __context += 1;
    e();
    __context -= 1;
  };
}

void a(void)
{b(); c();}

void main(void)
{a();}
```

Figure 5: Program from Fig. 2, after running `context` phase

Now, the programmer can specialize the function code, based on the value of `__context` (see Section 5).

4.4 Displaying the Context Graph

This project provides the `csg` phase that, given a C program, will print (to standard out) the DOT notation for the program's annotated `ContextGraph`. To execute this code on a file, use the following command:

```
cbz -csg <file>
```

The output from this phase can be piped to `dot` in the following way, to create a postscript file of the graph:

```
cbz -csg <file> | dot -Tps -o <file>.ps
```

Class `ContextViewer` is a C-Breeze walker that peforms the work of the `csg` phase. This phase simply invokes `ContextWalker`'s `buildContextGraph` method to obtain a `ContextGraph` object. Then the phrase invokes the `ContextGraph`'s `printDot` method. All the annotated graph diagrams in this report were created with the `csg` phase. Section 5 describes how the programmer can use such code to selectively execute statements based on context.

4.5 Performance Considerations

Tables 3 and 4 list the time complexity for graph operations. In these tables, n refers to the number of nodes, and m refers to the number of edges (callsites). Many operations (e.g., `addNode`, `context`) are $O(n)$ or $O(m)$, because the operations first search through the list of nodes or edges, to ensure correctness.

Operation	Complexity
`addNode(node)`	$O(n)$
`getNodes()`	$O(1)$
`name(node)`	$O(n)$
`name(node, string)`	$O(n)$
`addEdge(node, node, callNode *)`	$O(1)$
`deleteEdge(edge)`	$O(m)$
`getEdges()`	$O(1)$
`getPreds(node)`	$O(1)$
`getSuccs(node)`	$O(1)$
`label(edge)`	$O(m)$
`label(edge, int)`	$O(m)$
`getSource(edge)`	$O(1)$
`getTarget(edge)`	$O(1)$
`getCallNode(edge)`	$O(1)$
`topsort()`	$O(n+m)$
`findSCC()`	$O(n+m)$
`collapseSCC()`	$O(m*n)$
`printDot()`	$O(m)$

Table 3: `CallSiteGraph` performance

Operation	Complexity
`context(node)`	$O(n)$
`context(node, int)`	$O(n)$
`modifier(callNode *)`	$O(n)$
`modifier(callNode *, int)`	$O(n)$
`algorithm()`	$O(m*n)$

Table 4: `ContextGraph` performance

5 Selectively Executing Code

The programmer can use the two phases created by this project — `context` and `csg` — to modify the code to selectively execute statements based on context. The programmer first should run the `context` phase on the code, to obtain code that keeps track of the current context number. Then the programmer should use the graph produced by the `csg` phase to determine the context number(s) for the particular function in which context-sensitive code should execute. Then the programmer can insert `if` statements that compare the value of `__context` to the desired value. As a simple example, function `e` in Fig. 6 will print the string `''hello''` only when the function has been called by function `b`.

6 Future Work

In this section, we briefly discuss an alternate idea for handling recursion. The current solution handles recursion by collapsing the strongly connected components of the callsite graph. This technique is lossy, because we cannot know whether recursion occured or how many times it occurred. A reasonable expectation for recursion might be that we care only whether recursion has occured. Consider the callsite graph in Fig. 7. When at function `c`, we might want to know whether we have reached `c` from a recursive call to `b` or from a non-recursive call to `b`. In other words, we want to differentiate between two contexts: $\texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{c}$ and $\texttt{a} \rightarrow \texttt{b} \xrightarrow{+} \texttt{b} \rightarrow \texttt{c}$, where $\texttt{b} \xrightarrow{+} \texttt{b}$ means that `b` called itself one or more times.

This notation leads to the observation that contexts are really regular expressions. The callsite graph in Fig. 7 has the regular expression $\texttt{a}(\texttt{b}+)\texttt{c}$. This regular expression yields an infinite number of possible contexts. However, for our purposes, we only care about two contexts, described by the regular expression $\texttt{ab}(\texttt{b}?)\texttt{c}$. In other words, for simple recursion, we can reduce a regular expression that accepts an infinite number of strings to a regular expression that accepts a finite number of strings. The only remaining issue is to devise instrumentation that differentiates between the two contexts. The instrumentation must ensure that the first recursive call in a context is recorded and all subsequent recursive calls are idempotent with respect to the context.

While this technique works for simple recursion, more complex recursive cases require more study. Consider the example in Fig. 8. This callsite graph yields the regular expression $\texttt{a}((\texttt{b}(\texttt{cb})^*\texttt{c}?)|(\texttt{c}(\texttt{bc})^*\texttt{b}?))\texttt{d}$. However, we might reasonably expect to be concerned only with the contexts:

```
abd
acd
abcd
acbd
abcbd
acbcd
```

which can be enumerated by the regular expression $\texttt{a}((\texttt{b}((\texttt{cb})|\texttt{c})?)|(\texttt{c}((\texttt{bc})|\texttt{c})?))\texttt{d}$. The rule here for transforming infinite regular expressions to finite regular expressions

```
#include <stdio.h>

int __context = 0;

void f(void) {}

void e(void)
{
  if (__context == 1)
    printf("hello\n");
}

void d(void) {}

void c(void)
{e(); f();}

void b(void)
{
  d();
  {
    __context += 1;
    e();
    __context -= 1;
  };
}

void a(void)
{b(); c();}

void main(void)
{a();}
```

Figure 6: Example of selectively executed code

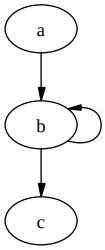

Figure 7: A callsite graph with a recursive function

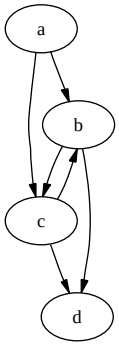

Figure 8: A callsite graph with mutually recursive functions

is more complex and would be a good starting point for future work.

In any case, to obtain more precision in the contexts we can recognize, the work on simple recursion outlined in this section could be applied with little effort to the current body of work.

7 Acknowledgements

I gratefully acknowlege Sam Guyer, Mike Bond, and Alison Norman for the feedback they gave me as I developed this project. I am particularly indebted to Mike Bond for the idea of adapting the Ball-Larus path-profiling algorithm to call graphs.

References

[1] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[2] E. Koutsofios and S. C. North. *Drawing graphs with dot*. Murray Hill, NJ.

[3] C. Lin, S. Z. Guyer, and D. Jimenez. The C-Breeze Compiler Infrastructure. TR 01-43, The University of Texas at Austin, Austin, TX, USA, 2001.