Precise Calling Context Encoding

William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, Xiangyu Zhang
Department of Computer Science, Purdue University
{wsumner,zheng16,dweeratu,xyzhang}@cs.purdue.edu

ABSTRACT

Calling contexts are very important for a wide range of applications such as profiling, debugging, and event logging. Most applications perform expensive stack walking to recover contexts. The resulting contexts are often explicitly represented as a sequence of call sites and hence bulky. We propose a technique to encode the current calling context of any point during an execution. In particular, an acyclic call path is encoded into one number through only integer additions. Recursive call paths are divided into acyclic subsequences and encoded independently. We leverage stack depth in a safe way to optimize encoding: if a calling context can be safely and uniquely identified by its stack depth, we do not perform encoding. We propose an algorithm to seamlessly fuse encoding and stack depth based identification. The algorithm is safe because different contexts are guaranteed to have different IDs. It also ensures contexts can be faithfully decoded. Our experiments show that our technique incurs negligible overhead (1.89% on average). For most medium-sized programs, it can encode all contexts with just one number. For large programs, we are able to encode most calling contexts to a few numbers.

1. INTRODUCTION

The goal of calling context encoding is to uniquely represent the current context of any execution point using a small number of integer identifiers (IDs), ideally just one. Such IDs are supposed to be automatically generated at runtime by program instrumentation. Efficient context encoding is important for a wide range of applications.

Event logging is essential to understanding runtime interactions between different components of large distributed or parallel systems. However, different modules in these systems tend to use the same library to communicate, e.g., sending a message using a socket library. Simply logging these communication events often fails to record the intents of these events. Recording their contexts would be very informative, but on the other hand, expensive and bulky, as it often implies walking stack frames to reconstruct a context and explicitly dumping the context as a sequence of symbolic function names. It has been shown in [22] that context sensitive event logging is critical for event reduction, which speeds up execution replay by removing events in a replay log that are not relevant to producing a failure. In that work, context information was retrieved through expensive stack walks. Calling contexts have also been used to reverse engineer the format of program input in AUTOFORMAT [11]. In aspect oriented programming, properties of calling contexts may be used to precisely locate aspects and have been used to support gathering execution information for debugging and unit test generation [19]. Context information has been shown to be very useful in testing sensor network applications in [9].

Context encoding can also improve bug reporting. The backtrace of a failure, itself a context, is a very useful component in a bug report. With context encoding, succinct bug reports can be generated. Moreover, it is also possible to collect contexts of additional execution points besides the failure point. For programs without symbolic information (for the sake of intellectual property protection), context encoding provides a way to anonymously represent contexts and allows them to be decoded at the developers' site.

Context sensitive profiling is very important to program optimization [23, 3, 5]. It annotates program profiles, such as execution frequencies, dependences, and object life times, with context information. Stack walking is too expensive when profile information is generated at a high frequency. Context sensitive optimizations [21, 7] often specify how programs should behave in various contexts to achieve efficiency. For example, region-based memory management [7] tries to cluster memory allocations into large chunks, called regions, so that they can be explicitly managed; context sensitive region-based memory management specifies in which region an allocation should be performed under various contexts. Such analyses need to disambiguate the different contexts reaching a program point at runtime to decide if the current context is one of those specified. Context encoding is highly desirable in this case.

Realizing the importance of context encoding, in recent years, a few encoding methods have been proposed. In [5], a technique is proposed to instrument call sites to cumulatively compute a hash of the function and line number containing the call site. The same encoding is guaranteed to be produced if the same context is encountered because the hash function is applied over the same data in the same order. The technique lacks a decoding component, meaning the context cannot be directly decoded from the computed hash. Note that such capability is essential to applications that require inspecting and understanding contexts. Moreover, different contexts may have the same encoding.

In [13], an approach is proposed to change stack frame sizes by allocating extra space on stack frames such that the stack offset, which is essentially the aggregation of stack frame sizes, disambiguates the contexts. This approach is not safe either, especially in the presence of recursion. The reason is that the number of stack frames at runtime could be arbitrary such that the aggregated size cannot be statically analyzed or safely profiled. Hence, the extra space on individual stack frames cannot be safely determined. The technique relies on offline training to generate a decoding dictionary. Both the inherent imprecision and incompleteness in the training set may lead to failure of decoding. The reported failure rate could be as high as 27% [13].

In this paper, we leverage the Ball-Larus (BL) control flow encoding algorithm to solve the context encoding problem. The scenario of encoding contexts has different constraints such that a more efficient algorithm can be devised. The basic idea is to instrument function calls with additions to an integer ID such that the value of the ID uniquely identifies contexts. Our algorithm is safe, uniquely identifying different contexts. It can precisely recover a context from its encoding. It has low overhead and handles function pointers, stack allocations, recursion, and so on.

Our main contributions are summarized as follows.

- We leverage the BL algorithm to encode acyclic contexts. The algorithm is more efficient as it exploits the unique characteristics of calling context encoding.
- We propose an algorithm to encode recursive contexts. Recursive contexts are divided into acyclic sub-sequences that are encoded independently. The sub-sequence encodings are stored to a stack. A featherweight generic compression further reduces the stack depth.
- We propose an algorithm to safely leverage stack depths to disambiguate contexts. Instead of allocating extra space on stack to distinguish contexts, we use our acyclic encoding algorithm in a lazy fashion, meaning that we apply it to contexts that cannot be safely disambiguated through stack offsets.
- We have a publicly available prototype implementation [1]. We evaluate it on a set of SPEC benchmarks and other large real world benchmarks. Our experiments show that our technique has very low overhead (1.89% on average) and it can encode all contexts of most medium-sized programs with just one 32 bits integer. For large programs, it can encode most runtime contexts with a few numbers.

2. MOTIVATION

```
1.  if (p1)
2.      s1;
3.  else
4.      s2;
5.  if (p2)
6.      s3;
7.  else
8.      s4;
```

path	id
1256	0
1258	1
1456	2
1458	3

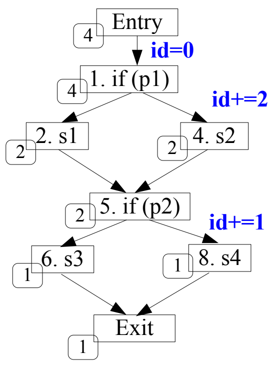


Figure 1: Example for Ball-Larus path encoding. Instrumentation is marked on control flow edges. Node annotations (rounded boxes at the corners) represent the number of paths starting from the annotated point.

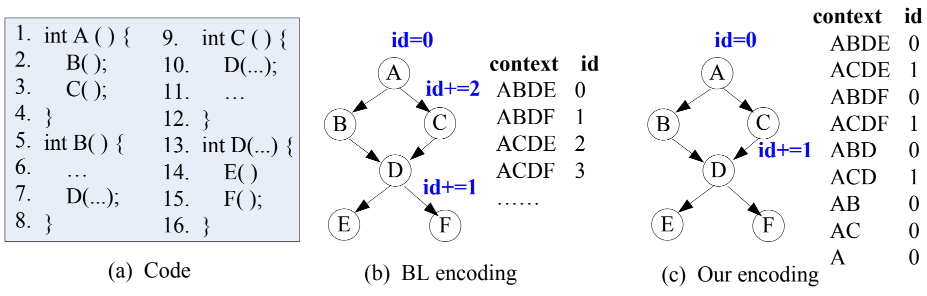
Background: Ball-Larus Path Encoding. In the seminal paper [4], Ball and Larus proposed an efficient algorithm (referred to as the BL algorithm) to encode intra-procedural control flow paths taken during execution. The basic BL algorithm translates an acyclic path encoding into instrumentation on control flow edges. At runtime, a sequence of instrumentation is executed following a control flow path, resulting in the path identifier being computed. All instrumentation involves only simple additions. The idea can be illustrated by the example in Fig. 1. The code is shown on the left and the control flow graph is shown on the right. Instrumentation is marked on control flow edges. Before the first statement, *id* is initialized to 0. If the false branch is taken at line 1, *id* is incremented by 2. If the false branch is taken at line 5, *id* is incremented by 1. As shown on bottom left, executions taking different paths lead to different values in *id*. Simply, *id* encodes the path.

The algorithm first computes the number of paths leading from a node to the end of the procedure. For example, node 1 has four paths reaching `Exit`. Such numbers are annotated as rounded boxes in Fig. 1. Given a node with n paths, the instrumentation from the node to `Exit` generates IDs falling into the range of $[0, n)$ to denote the paths. For instance, the instrumentation in Fig. 1 generates paths with IDs in [0,4). In particular, the instrumentation on $1 \rightarrow 4$ separates the range into two sub-ranges: [0,2) and [2,4), denoting the paths following edges $1 \rightarrow 2$ and $1 \rightarrow 4$, respectively. The instrumentation on $5 \rightarrow 8$ further separates the two paths from 5 to `Exit`.

More details about the BL algorithm can be found in [4]. The algorithm has become canonical in control flow encoding and been widely used in many applications [10, 6, 16].

Inadequacy of BL for Context Encoding. Although the BL algorithm is very efficient at encoding control flow paths, we observe that it is inadequate for encoding calling contexts, which are essentially paths in a call graph. Consider the example in Fig. 2. The code is shown on the left. Figures (b) and (c) show the call graph with two encoding schemes. The BL encoding is presented in (b), and (c) shows more efficient encoding. Nodes in a call graph represent functions and edges represent function calls. BL encoding is designed to encode statement granularity paths leading from the entry of a function to the end of it. The criterion is that each of these paths has a unique encoding. As shown in Figure (b), all the paths leading from `A` to `E` or `F` have different encodings. However, context encoding has a different criterion, that is, *all unique paths leading from the root to a specific node have unique encodings, because we only need to distinguish the different contexts with respect to that node.* In other words, there are no constraints between the encodings of paths that end at different nodes. It is fine if two paths that end at different nodes have the same encoding. For example, paths `ABDE` and `ABDF` have the same encoding 0 according to the scheme in (c). As a result, although the encoding in Figure (c) has less instrumentation (on 2 edges versus 3 in (b)) and requires a smaller encoding space (the maximum ID is 1 versus 3 in (b)), it still clearly distinguishes the various contexts of any node.

Our technique is based on the above observation. It handles recursive calls and function pointers. It also leverages stack offset, namely, the offset of the stack pointer regarding the stack base, to achieve efficiency. More important, our



```
int A ( ) {          int C ( ) {
    B( );                D(...);
    C( );                ...
}                    }
int B( ) {           int D(...) {
    ...                  E( )
    D(...);              F( );
}                    }
```

(a) Code

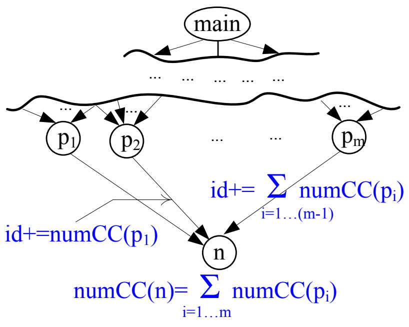
context	id
ABDE	0
ABDF	1
ACDE	2
ACDF	3
……	

(b) BL encoding

context	id
ABDE	0
ACDE	1
ABDF	0
ACDF	1
ABD	0
ACD	1
AB	0
AC	0
A	0

(c) Our encoding

Figure 2: The inadequacy of the BL algorithm for encoding contexts.



main
p_1
p_2
p_m
n
$id{+}{=}\sum_{i=1\ldots(m-1)} numCC(p_i)$
$id{+}{=}numCC(p_1)$
$numCC(n)=\sum_{i=1\ldots m} numCC(p_i)$

Figure 3: Intuition of our algorithm.

algorithm is precise, meaning it ensures that each context has a unique encoding, and it allows decoding.

3. DEFINITIONS

Definition 1. *A call graph (CG) is a pair $\langle N, E\rangle$. N is a set of nodes with each node representing a function. E is a set of directed edges. Each edge $e \in E$ is a triple $\langle n, m, \ell\rangle$, in which $n, m \in N$, represent a caller and callee, respectively, and ℓ represents a call site where n calls m.*

In the above definition of call graph, call edges are modeled as a triple instead of a caller and callee pair because we want to model cases in which a caller may have multiple invocations of the callee.

Definition 2. *The calling context (CC) of a given function invocation m, is a path in the CG leading from the root node to the node representing m.*

The context of an execution point can be computed by concatenating the context of its enclosing function invocation and the program counter (PC) of the point.

Definition 3. *A valid calling context encoding scheme is a function $En : CC \rightarrow \overline{Z}$ such that*

$$\forall n \in N, \forall x, y \in \{\text{the CCs of } n\} \wedge x \neq y, En(x) \neq En(y)$$

Any encoding scheme that generates unique encodings, i.e., integer sequences ($\overline{Z}$ represents a sequence of integers), for unique contexts of the same function is a valid encoding scheme. For example, a naïve but valid encoding scheme is to use the sequence of call site PCs to denote a context.

Our research challenge is to devise a highly efficient valid encoding scheme which also allows precise decoding. Fig. 2 (c) presents an example of such a scheme.

4. ENCODING ACYCLIC GRAPHS

In this section, we introduce an algorithm to encode calling contexts that do not involve recursion. The basic idea is illustrated in Fig. 3. Assume function $numCC(n)$ represents the number of contexts of a node n such that $numCC(n) = \sum_{i=1\ldots m} numCC(p_i)$ where for $i \in [1, m]$, p_i are the parents of n. A critical invariant of our technique is that *the $numCC(n)$ contexts of n should be encoded by the numbers in the range of* $[0, numCC(n))$. To do so, the edge instrumentation should separate the range into m disjoint subranges, with $[0, numCC(p_1))$ representing the $numCC(p_1)$ contexts along edge $p_1 \rightarrow n$, $[numCC(p_1), numCC(p_1) + numCC(p_2))$ representing the $numCC(p_2)$ contexts along $p_2 \rightarrow n$, and $[\sum_{j=1\ldots(i-1)} numCC(p_j), \sum_{j=1\ldots i} numCC(p_j))$ encoding the $numCC(p_i)$ paths along $p_i \rightarrow n$. As shown in Fig. 3, this can be achieved by instrumenting an edge $p_i \rightarrow n$ with "$id{+} = \sum_{j=1\ldots(i-1)} numCC(p_j)$".

Algorithm 1 Encoding for Acyclic CGs.

```
1:  Annotate (N, E) {
2:    for $n \in N$ in topological order do:
3:      for each parent $p$ of $n$ do:
4:        $numCC[n] \leftarrow numCC[n] + numCC[p]$
5:  }
6:  Instrument (N, E) {
7:    Annotate (N,E)
8:    for $n \in N$ do:
9:      $s \leftarrow 0$
10:     for each $e = \langle p, n, \ell\rangle$ in $E$ do:
11:       annotate $e$ with "$+s$"
12:       insert $\boxed{id = id + s}$ before $\ell$
13:       insert $\boxed{id = id - s}$ after $\ell$
14:       $s \leftarrow s + numCC[p]$
15: }
```

The algorithm is presented in Algorithm 1. It first computes the number of calling contexts for each node (stored in $numCC$). It then traverses each node n in the main loop in lines 8-14. For each edge $e = \langle p, n, \ell\rangle$, the following instrumentation is added: before the invocation at ℓ, the context identifier id is incremented by the sum s of the $numCC$s of all preceding callers; after the invocation, id is decremented by the same amount to restore its original value.

Consider the example in Fig. 4. Node annotations (i.e. numbers in boxes) are first computed. In the first three steps of the topological traversal, nodes `A`, `B`, and `J` are annotated with 1, meaning these functions have only one context; node `D`'s annotation is the sum of those of `B` and `J`, denoting there are two possible contexts when `D` is called; the remaining nodes are similarly annotated. The program is instrumented based on the annotations. Consider the invocations to `I`, which are from `F`, `G`, and `J`. The empty label on edge `FI` means that the invocation is instrumented with "$id{+} = 0$" and "$id{-} = 0$" before and after the call, respectively. This instrumentation is optimized away. The edge `GI` has the label "+4", meaning the instrumentation before and after comprises "$id{+} = 4$" and "$id{-} = 4$". Note that 4 is the annotation of `F`. Similarly, since the sum of the annotations of `F` and `G` are 4+3=7, edge `JI` has the label "+7". Other edges are similarly instrumented. At runtime, the instrumentation yields the encodings as shown in the table in

context	id
AB	0
AJ	0
ABD	0
AJD	1
ABE	0
ABD^{1}F	0
AJD^{1}F	1
ABD^{2}F	2
AJD^{2}F	3
…	

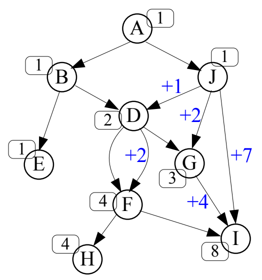
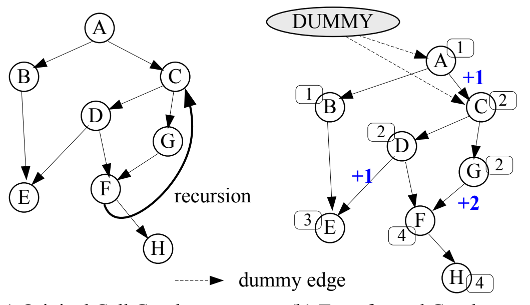


Figure 4: Example for acyclic encoding. Label “$+c$” means “$id+=c$” is added before the invocation and “$id-=c$ is added after; superscript on D is to disambiguate call sites.



(a) Original Call Graph (b) Transformed Graph

```
void A (…) {        void F (…) {
   B (…);              while (…) {
   …                      push(<id,l>);
   id+=1;                 id=0;
   C (…);            l:   C (…);
   id-=1;                 id=pop().first;
   …                      …
}                      }
                       H (…);
                       …
                    }
```

(c) Instrumentation

Figure 5: Example for encoding cyclic CGs.

Fig. 4.

In applications such as context sensitive profiling and context sensitive logging, context IDs are emitted as part of the profile. In order to facilitate human inspection, decoding is often needed. The decoding algorithm is presented in Algorithm 2. The algorithm traverses from the given function in a bottom-up fashion and recovers the context by comparing the encoding with edge annotations. In particular, at line 2, the recovered context cc is initialized with the given function m, which is the leaf node of the final context. Lines 4-10 compose the main process, which terminates when the the root node is reached. In the inner loop from lines 5 to 9, the algorithm traverses edges ending at the current function n. At line 6, it tests if the encoding falls in the encoding range of the contexts along the current edge. If so, the edge is taken; the caller p and the call site ℓ are attached to the recovered context at line 7. Symbol ‘$\bullet$’ represents concatenation. At line 8, the encoding is updated by subtracting the edge annotation. This essentially reverses one step of encoding. The process continues by treating the caller as the new current function at line 10.

Consider the example in Fig. 4. Assume the ID 6 is generated at function I. The algorithm starts from I. Since $En(\texttt{GI}) \equiv 4 \leq 6 < 7 \equiv En(\texttt{GI}) + numCC(\texttt{G})$, the context must have followed the edge GI. The edge is taken and the encoding is decremented by 4 to the value 2. At G, since $En(\texttt{JG}) \equiv 2 \leq 2 < 3 \equiv En(\texttt{JG}) + numCC(J)$, edge JG is taken. Finally, edge AJ is taken, yielding the context AJGI.

Algorithm 2 Decode a context Id.

Input: the encoding id; the function m at which the encoding was emitted; the edge set E; the edge annotations En.
Output: the explicit context cc.

```
1:  Decode ($id$, $m$, $E$, $En$) {
2:      $cc \leftarrow$“$m$”
3:      $n \leftarrow m$
4:      while $n \neq root$ do:
5:          for each $e = \langle p, n, \ell \rangle$ in $E$ do:
6:              if $En(e) \leq id < En(e) + numCC[p]$ then:
7:                  $cc \leftarrow$“$p^{\ell}$” $\bullet$ $cc$
8:                  $id \leftarrow id$ - $En(e)$
9:                  break
10:         $n \leftarrow p$
11: }
```

5. ENCODING WITH RECURSION

In the presence of recursion, a context may be of unbounded length, making encoding using a bounded number infeasible. We propose to use a stack to encode contexts with recursion. The basic idea is to encode the acyclic sub-paths of a recursive context. When a recursion occurs, the current acyclic encoding is pushed to the stack and the following acyclic sub-path is encoded with a new id. The numbers on stack and the current ID together represent the context. In order to perform correct acyclic sub-path encoding, recursive CGs need to be transformed.

Our solution is presented in Algorithm 3. The first step deals with CG transformation (in *AnnotateRecursive()*). A dummy root node is introduced. A dummy edge is introduced between the new root and the original root. Dummy edges are further introduced between the new root and any nodes that are the target of a back edge. Note that only one dummy edge is introduced even if there are multiple back edges to the same node. A dummy edge is always the first edge in the edge set of a node. In the transformed graph, back edges are removed to allow acyclic encoding. Consider the example in Fig. 5. The recursion FC is removed and dummy edges are introduced between the dummy root and the original root A, as well as between the dummy root and the recursive edge target C. Intuitively, after transformation, acyclic sub-sequences of a recursive context become valid contexts in the transformed CG. Hence, they can be taken into account in the annotation computation. In Fig. 5 (b), the dummy edge from DUMMY to C makes the acyclic sub-paths, such as CDF and CGF, become valid contexts and have unique encodings in the transformed graph. Note that paths that do not involve recursion, such as ACGFH, are not divided into sub-paths, even if they contain a node that may be the target of a back edge, such as C in this case.

The instrumentation algorithm is shown in function *InstrumentRecursive()*. The instrumentation is performed on the original graph, which may have back edges. Since the transformed graph shares the same node set as the original graph (except the dummy root), the acyclic node annotations on the transformed graph are also annotations on the original graph and hence used in instrumentation. Similar to the previous algorithm, s maintains the sum of contexts of callers that precede the one being processed. At line 12, it is initialized to 1 if the node could be a back edge target, 0 otherwise. Setting to 1 respects the numbering caused by the dummy edge on the transformed graph. Lines 14-17 handle non-back-edges, and they are the same for acyclic graphs. Lines 19-21 handle back edges. Specifically, before a recursive invocation, the current *id* and the call site are pushed to the stack, and *id* is reset to 0. Resetting *id* indicates that the algorithm starts to encode the next acyclic sub-path. After the invocation, *id* is restored to its previous value. Fig. 5 (c) shows the instrumentation for functions A and F, which are the callers of C.

Algorithm 3 Handling Recursion.

Description:
$\langle N', E'\rangle$ represents the transformed CG;
stack is the encoding stack.

```
1:  AnnotateRecursive ($N$, $E$) {
2:      $N' \leftarrow \{\texttt{DUMMY}\} \cup N$
3:      $E' \leftarrow E \cup \{\langle \texttt{DUMMY}, root, -\rangle\}$
4:      for each back edge $e = \langle p, n, \ell\rangle$ in $E$ do:
5:          $E' \leftarrow E'$ - $e$
6:          $E' \leftarrow E' \cup \{\langle \texttt{DUMMY}, n, -\rangle\}$
7:      Annotate($N'$, $E'$)
8:  }
9:  InstrumentRecursive ($N$, $E$) {
10:     AnnotateRecursive ($N$, $E$)
11:     for $n \in N$ do:
12:         $s \leftarrow$ ($n$ has a dummy edge in $E'$) ? 1 : 0
13:         for each edge $e = \langle p, n, \ell\rangle$ in $E$ do:
14:             if $e$ is not a back edge then:
15:                 insert $\boxed{id = id + s}$ before $\ell$
16:                 insert $\boxed{id = id - s}$ after $\ell$
17:                 $s \leftarrow s + numCC(p)$
18:             else:
19:                 insert $\boxed{\text{push}(\langle id, \ell\rangle)}$ before $\ell$
20:                 insert $\boxed{id = 0}$ before $\ell$
21:                 insert $\boxed{id = \text{pop}().first}$ after $\ell$
22: }
23: DecodeStack ($id$, $stack$, $m$, $E'$, $En^{E'}$) {
24:     $\ell \leftarrow \emptyset$
25:     while true do:
26:         Decode ($id$, $m$, $E'$, $En^{E'}$)
27:         $wcc \leftarrow cc \bullet wcc$
28:         if $stack.empty()$ then:
29:             break
30:         $\langle id, \ell\rangle \leftarrow stack.pop()$
31:         $m \leftarrow$ the residence function of $\ell$
32: }
```

Consider the example context ACGFCDE. It is encoded as a sub-path with ID 3 on the stack and the current sub-path $id = 1$. The encoding 3 is pushed to the stack before F calls C. After the *id* value is pushed, it is reset to 0. As a result, taking the remaining path CDE leads to $id = 1$. Assume the execution returns from E, D, C, and then calls C, D, F, and H, yielding the context of ACGFCDFH. The context is encoded as 3 on the stack and current $id = 0$.

The decoding algorithm for recursive CGs is presented in function *DecodeStack()*. It takes the numbers on the stack, the current sub-path encoding, and the current function as part of its inputs. Intuitively, it decodes one acyclic sub-path of the recursive context at a time, until all encodings on the stack are decoded. Decoding an acyclic sub-path is done by calling the acyclic decoding algorithm on the transformed graph at line 26. The resulting acyclic sub-path *cc* is concatenated with the whole recovered context *wcc*. At line 31, the call site label ℓ is used to identify the method in which the *id* was pushed to the stack, which is also the starting point of the next round of decoding.

Consider an example. Assume we want to decode a context represented by the stack having an entry $\langle 3, \texttt{F to C}\rangle$, the current $id = 1$, and the current function E. After the first iteration, i.e. $Decode(1, \texttt{E}, ...)$, the sub-path CDE is decoded. Function F is decided as the starting point of the next round of decoding, according to the call site on the stack. After the round, the value 3 on the stack is decoded to the sub-path ACGF. The two sub-paths constitute the context.

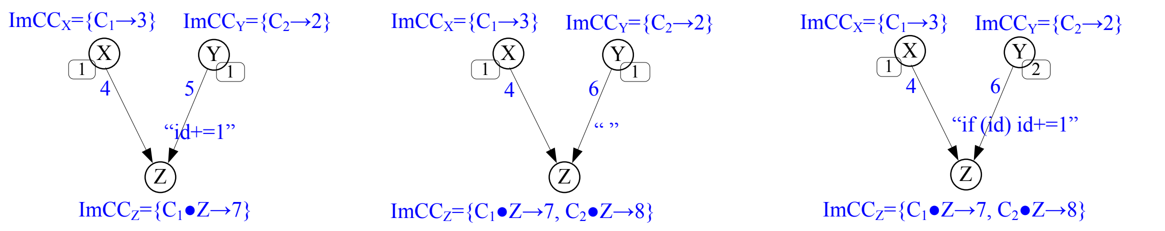
Our design can easily handle very deep contexts caused by highly repetitive recursive calls. In such cases, contexts are a string with repetitive substring patterns such as ACGF CGF CGF CGF ... in Fig. 5. Such redundancy is not directly removable without relatively expensive compression. With encoding, the repetitive patterns are encoded into repetitive integers and can be further encoded as a pair of ID and frequency. The above context can be encoded to two pairs 3 : 1 and 2 : 3, with the former representing ACGF and the latter representing the three repetitions of CGF.

6. SAFE HYBRID ENCODING LEVERAGING STACK OFFSETS

The encoding algorithms we have discussed so far explicitly produce a unique id for each calling context. We call them *explicit encoding* techniques. At runtime, it is often the case that the stack offset, namely, the value of the current stack pointer subtracted by the base of the entire stack, can disambiguate the current context. Consider the example in Fig. 2 (c). Previously, updates to *id* had to be inserted on edge CD to distinguish the two contexts of D. Let the stack offset at the entry of D in the context of ABD be x and the offset in the context of ACD be y. If x does not equal y, the two contexts can be disambiguated without any explicit encoding. We call such stack offset based encoding *implicit encoding* as explicit instrumentation is not needed.

In reality, a few factors make applying implicit encoding difficult. First of all, there may be multiple contexts that alias to the same implicit encoding, i.e., they have the same stack offset. Consider the example in Fig. 4. The two contexts $\texttt{ABD}^1\texttt{F}$ and $\texttt{ABD}^2\texttt{F}$ may have the same stack offset because the same sequence of functions are called. Second, programming languages such as C/C++ allow declaring variable-size local arrays. GCC allocates such arrays on stack. Stack allocations make stack offsets variable, and hence implicit encoding is infeasible. Third, in the presence of recursion, stack offsets cannot be statically reasoned about, which makes it inapplicable.

In order to address the aforementioned issues, we propose a hybrid encoding algorithm that performs explicit encoding when implicit encoding is not applicable. The algorithm is safe, meaning that it uniquely encodes each possible context. The intuition of the hybrid algorithm is presented in Fig. 6. Besides the number of contexts, each node is also annotated with a set of implicit contexts, denoted as $ImCC$, which represents a set of contexts of the node having distinct and fixed stack offsets. It is a mapping from contexts to their offsets. For instance, the implicit context set of node X in Fig. 6 contains context $\texttt{C}_1$ and its stack offset 3 (symbol $\mapsto$ represents the maps-to relation). Each edge is annotated with two pieces of information. The first piece is the stack frame offset, which is the difference between the stack pointer and the base of the current stack frame when the invocation denoted by the edge occurs. The stack offset of a context can

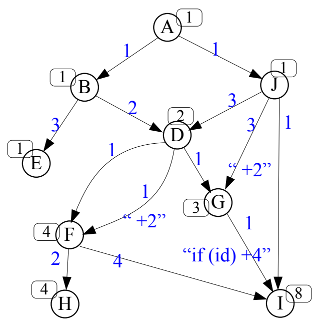


(a) 3+4==2+5 (b) 3+4 != 2+6 && |ImCC$_{X/Y}$|==numCC(X/Y) (c) 3+4 != 2+6 && |ImCC$_Y$|!=numCC(Y)

Figure 6: Intuition of Hybrid Encoding.

node	ImCC	
	context	offset
D	ABD	3
	AJD	4
F	ABD^{1}F	4
	AJD^{1}F	5
G	ABDG	4
	AJDG	5
I	ABD^{1}FI	8
	AJD^{1}FI	9
	ABDGI	5
	AJDGI	6
	AJI	2

(a) Implicit contexts



(b) New encoding scheme

context	offset	id
ABD^{1}FI	8	0
AJD^{1}FI	9	0
ABD^{2}FI	8	2
AJD^{2}FI	9	2
ABDGI	5	0
AJDGI	6	0
AJGI	5	6
AJI	2	0
ABDG	4	0
AJDG	5	0
AJG	4	2

(c) Context encodings

Figure 7: Example of Hybrid Encoding. Edge instrumentations are quoted and stack frame offsets are not.

be computed by aggregating the edge offsets along the context. The second annotation is the instrumentation, which is quoted. Symbol '•' represents concatenation.

The figure presents three cases. In case (a), the two implicit contexts of Z have a conflicting offset, namely, $3+4 \equiv 2+5$. We cannot implicitly encode both. In such a case, we implicitly encode the context along edge XZ and explicitly encode that from YZ. Hence, $ImCC_Z$ is set to contain the context from edge XZ and *id* is increased by $numCC(X) = 1$ along edge YZ. The instrumentation has the same effect of separating the encoding space as in previous algorithms. In case (b), the two implicit contexts do not conflict and all contexts of X and Y can be implicitly encoded, implied by the sub-condition $|ImCC_{X/Y}| \equiv numCC(X/Y)$. In such a case, no explicit encoding is needed and the *ImCC* set of Z contains both. In case (c), the two implicit contexts of Z do not conflict but the contexts of Y are heterogeneously encoded, denoted by $|ImCC_Y| \neq numCC(Y)$. It implies that *id* may be non-zero at Y, depending on the context at runtime. If *id* is not zero, explicit encoding must have been used, and the new context of Z should be explicitly encoded. Hence, *id* is increased by $numCC(X) = 1$. If *id* is zero, the context of Y is one of the contexts in $ImCC_Y$. Because the corresponding context of Z does not have conflicts and can be implicitly encoded, no update to *id* is needed. The above logic is realized by the guarded instrumentation on edge YZ in Fig. 6 (c).

The algorithm is presented in Algorithm 4. Variable *ImCC* represents the set of contexts that are implicitly encoded for each node. Function *extend*($ImCC[p]$, n) extends the implicit contexts in $ImCC[p]$ along the invocation $p \rightarrow n$. This is done by concatenating n to the contexts in $ImCC[p]$ and increasing the corresponding stack offsets by the stack frame offset of $p \rightarrow n$. Function *conflict*($ImCC_x$, $ImCC_y$) tests if two implicit context sets conflict. This is done by checking if there are two contexts in the two respective sets alias to the same stack offset.

The instrumentation algorithm is described in function *Instrument()*. Here we only present the algorithm for acyclic graphs. The extension to recursion can be achieved in a way similar to explicit encoding and hence omitted. The algorithm traverses the nodes in the topological order. It first resets the *ImCC* for the current node n at line 3. It then traverses the set of invocations to n, denoted by edges of the form $\langle p, n, \ell \rangle$, with p the caller and ℓ the call site. It extends $ImCC[p]$ to n at line 6, and then tests if the extended set conflicts with the implicit contexts of n that have been computed so far. If there is no conflict, the extended set is admitted and aggregated to the implicit set of n at line 8. Line 9 checks if all contexts of p are implicit. If so, the aforementioned case (b) in Fig. 6 is encountered. There is no need to instrument the invocation. If not, case (c) is encountered, namely, the contexts of p may be explicitly encoded. Hence, the instrumentation should decide at runtime if the context has been explicitly encoded. If so, the instrumentation will continue to perform explicit encoding as shown by the boxes in lines 11-12. If the extended implicit set incurs conflict, case (a) is encountered. In lines 15-16, the edge is explicitly encoded.

Encoding Example. Consider the earlier example in Fig. 4. Fig. 7 (b) shows the graph with stack frame offset annotations and instrumentation (quoted). For instance, when B is called inside A, the stack frame offset is 1; when D is called inside B, the frame offset is 2. Hence, the stack offset of the context ABD is the sum of the two, which is 3. Similarly, the stack offset of AJD is 1+3=4. The two different offsets disambiguate the two contexts, and thus the implicit context

Algorithm 4 Hybrid Encoding.

DEFINITIONS:
ImCC[ ] an array of implicit contexts indexed by nodes;
extend (*ImCC*[*p*], *n*)=

$$\{C \bullet n \;\mapsto\; t + \mathit{offset}(p, n) \mid C \mapsto t \in \mathit{ImCC}[p]\}$$

in which *offset*(*p*, *n*) is the stack frame offset of the call to *n* in *p*.
conflict (*ImCC*$_x$, *ImCC*$_y$)=

$$\begin{cases} 1 & \exists\, C_1 \mapsto t \in \mathit{ImCC}_x \wedge C_2 \mapsto t \in \mathit{ImCC}_y \\ 0 & \mathit{otherwise} \end{cases}$$

```
1: Instrument (N, E) {
2:   for $n \in N$ in topological order do:
3:     $ImCC[n] \leftarrow \emptyset$
4:     $s \leftarrow 0$
5:     for each edge $e = \langle p, n, \ell \rangle$ in $E$ do:
6:       $ImCC' \leftarrow extend\ (ImCC[p], n)$
7:       if not conflict ($ImCC[n]$, $ImCC'$) then:
8:         $ImCC[n] \leftarrow ImCC[n] \cup ImCC'$
9:         if $|ImCC[p]| \neq numCC[p]$) then:
10:          /*case (c) in Fig. 6, otherwise case (b)*/
11:          replace the call $\boxed{\ell :\quad n(\ldots)}$ in $p$ with
12:            if (id) {
                 id = id + s;
                 n(...);
                 id = id − s;
               } else n(...);
13:      else:
14:        /*case (a) */
15:        insert $\boxed{id = id + s}$ before $\ell$
16:        insert $\boxed{id = id - s}$ after $\ell$
17:      $s \leftarrow s + numCC[p]$
18: }
```

set of D, as shown in Fig. 7 (a), contains both contexts. No instrumentation is needed.

Now let us consider F. When the first edge is processed. *extend*(*ImCC*[D^1], F) is computed as $\{\mathtt{ABD}^1\mathtt{F} \mapsto 4,\ \mathtt{AJD}^1\mathtt{F} \mapsto 5\}$ at line 6, and it is assigned to *ImCC*[F] at line 8. When the second edge is processed, *extend*(*ImCC*[D^2], F) is computed as $\{\mathtt{ABD}^2\mathtt{F} \mapsto 4,\ \mathtt{AJD}^2\mathtt{F} \mapsto 5\}$. The conflict test at line 7 fails, so *ImCC*(F) only contains the set extended along the edge D^{1}F, as shown in Fig. 7 (a). Moreover, the algorithm instruments the edge D^{2}F according to lines 15-16.

When I is considered, the extensions from *ImCC*[F], *ImCC*[G] and *ImCC*[J] do not conflict. However, contexts to G may be explicitly encoded as $|ImCC[\mathtt{G}]| = 2 \neq 4 = numCC[G]$. The instrumentation has to be guarded as shown on the edge GI. Sample encodings can be found in Fig. 7 (c). Compared to the instrumentation in Fig. 4, in which 5 edges need to be instrumented with each instrumentation comprising one addition and one subtraction (2 reads and 2 writes), the hybrid version instruments 3 edges. Furthermore, the instrumentation on edge GI may need just one read (the read in the predicate).

Decoding Example. The decoding algorithm is elided for brevity. We will use examples to intuitively explain the idea. The encoding of a context under the hybrid algorithm is a triple, which comprises the stack offset, the explicit ID, and the current function. Note that only the explicit ID is computed by instrumentation, the other two can be inferred at any execution point. Assume we are given the encoding of $offset = 5$, $id = 0$ and the current function G. The explicit encoding $id = 0$ means that the context is not explicitly encoded and can be looked up from the *ImCC* set. From *ImCC*[G], we recover the context as AJDG.

Assume we are given the encoding $offset = 5$, $id = 6$ and the current function I. The nonzero explicit encoding means that explicit encoding is used. From the encoding graph in Fig. 7 (b), we know that explicit IDs at I in range [4, 7) represent contexts along the edge GI. We reverse both the stack offset and the explicit encoding along this edge and get $offset = 5 - 1 = 4$ and $id = 6 - 4 = 2$. The IDs at G in range [2,3) represent contexts along JG. Backtracking along the edge leads to $offset = 4 - 3 = 1$ and $id = 2 - 2 = 0$. Now with $id = 0$, we know that the remaining part of the context can be looked up, yielding AJ. Here we recover the whole context AJGI.

7. HANDLING PRACTICAL ISSUES

Handling Insufficient Encoding Space. Since our technique uses a 32 bit ID, it allows a function to have a maximum of 2^{32} different contexts. We observe for some large programs a function may have more than 2^{32} contexts. For example, in GCC, there are a few functions that are called by a few thousand other functions, leading to an overflow in the encoding space. In order to handle such cases, we propose a selective reduction approach. We use profiles to identify hot and cold call edges. Cold edges are replaced with dummy edges such that sub-paths starting with these cold edges can be separately encoded. As a result, the overall encoding pressure is reduced. At runtime, extra pushes and pops are needed when the selected cold edges are taken.

Handling Function Pointers. If function pointers are used, points-to analysis is needed to identify the targets of invocations. Due to the conservative nature of points-to analysis, the possible targets may be many. We employ a simple and safe solution. We profile the set of targets for a function pointer invocation with a number of runs. Edges are introduced to represent these profiled targets and then instrumented normally. During real executions, if a callee is out of the profiled set, push and pop are used.

Handling setjmp/longjmp. Setjmp allows a developer to mark a position in the calling context that a successive use of longjmp can then automatically return to, unwinding the calling context to the marked point. Our approach safely handles such unwinding by detecting when a setjmp is encountered and storing a copy of the context stack height and current context identifier within the local variables of the function containing setjmp. When longjmp unwinds the calling context, these values are then safely restored from the local copies, unwinding the context encoding as well.

Handling Stack Allocations. Implicit encoding is not possible when stack allocation is used, e.g., when allocating a variable-size array on the stack. We use static analysis to identify all functions that have stack local allocation and prohibit implicit encoding for those functions.

8. EVALUATION

We have implemented our approach in OCaml using the CIL source-to-source instrumentation infrastructure [14]. The implementation has been made available on our project website [1]. All experiments were performed on an Intel Core 2 2.1GHz machine with 2GB RAM and Ubuntu 9.04.

Note that because we use source level analyses for our implementation, we do not have as much information as would be available were the analysis within the actual compilation phase. In particular, we don't know if the compiler eventually inlines a function, resulting in indistinguishable stack frame sizes. Hence, we disabled inlining during our experiments. Observe that this is not a limitation of our algorithm but rather an outcome of our infrastructure selection. An implementation inside a compiler after functions are inlined can easily handle the issue.

Table 1 presents the static characteristics of the programs we use. Since CIL only supports C programs, our implementation currently supports C programs. We use SPECint 2000 benchmarks and a set of open source programs. Some of them are large, such as `176.gcc`, `alpine`, and `vim`. Three SPECint programs, `252.eon`, `254.gap` and `253.perlbmk` are not included because CIL failed to compile them due to their use of C++ or unsupported types. For each program, the table also details lines of code (LOC), the number of nodes in the call graph (CG nodes), the number of edges in the call graph (CG edges), the number of recursive calls or back edges in the call graph (recursions), the number of function pointer invocations, and the maximum ID required in the call graph under the original BL numbering scheme (BL max ID) and our scheme (our max ID).

We can observe that most programs make use of recursion and function pointers. In particular, `176.gcc` has 1800 recursive invocations and uses function pointers at 128 places. We are able to encode the contexts of all programs smaller than 100K LOC in a 32-bit ID. For the larger programs, overflows are observed and selective reduction (Section 7) is performed to reduce encoding pressure and fit the ID into 32 bits. The numbers below the table show the number of nodes on which selective reduction is performed for each large programs. Observe, the maximum ID for the original BL encoding scheme is often a few times larger than ours.

Fig. 8 illustrates the runtime overhead of our encoding algorithms. We use reference inputs for SPEC programs and random inputs for the rest. In particular, we use training inputs to profile SPEC programs. The runtime for `alpine 2.0` cannot be accurately measured as it is an interactive program. The slow down is not humanly observable though. The times for `vim` were collected in batch mode. We normalize the runtime of two encoding algorithms: one is the basic algorithm that handles recursion and the other the hybrid algorithm. From the figure, we observe that our technique is very efficient, the average overheads are 3.64% and 1.89% for the two respective algorithms. The hybrid algorithm improves over the basic algorithm by 48.1%. `176.gcc` and `255.vortex` have relatively higher overhead due to the extra pushes and pops caused by selective reduction.

Table 2 presents the dynamic properties. The second and third columns compare our encoding stack depth (ours) to the plain calling stack depth (plain), i.e., the call path length. The third and fourth columns show the stack size needed to cover 90% of contexts at runtime. The last column presents the number of unique contexts encountered. We observe that our maximum encoding stacks are substantially shorter than the corresponding maximum plain stacks. For most utility programs, the encoding stack is empty, meaning the contexts can be encoded into one ID without using the stack. We also observe that some programs need maximum encoding stacks with a non-trivial depth, e.g. `176.gcc`, `181.mcf`, `186.crafty` and `197.parser`. However, when we look at the 90% cutoffs, `176.gcc` and `181.mcf` require en-

programs	Max Depth		90% Depth		dynamic contexts
	ours	plain	ours	plain	
cmp 2.8.7	0	3	0	3	9
diff 2.8.7	0	7	0	5	34
sdiff 2.8.7	0	5	0	4	44
find 4.4.0	2	12	1	12	186
locate 4.4.0	0	9	0	9	65
grep 2.5.4	0	11	0	8	117
tar 1.16	3	40	2	31	1346
make-3.80	6	82	3	43	1789
alpine 2.0	11	29	6	18	7575
vim 6.0	10	31	5	10	3226
164.gzip	0	9	0	7	258
175.vpr	0	9	0	6	1553
176.gcc	19	136	2	15	169090
181.mcf	14	42	0	2	12920
186.crafty	34	41	10	23	27103471
197.parser	36	73	11	28	3023011
255.vortex	7	43	2	12	205004
256.bzip2	1	8	0	8	96
300.twolf	4	11	0	5	971
Average	8.78	39.22	2.17	13.72	1795292

Table 2: Dynamic context characteristics.

coding stacks of depth 2 and 0, respectively. To precisely evaluate our technique, we also contrast the full frequency distributions for our encoding stacks and the corresponding plain stacks. Due to space limit, we only present some of the results in Fig. 9. Each diagram corresponds to one of the benchmarks considered. The x-axis corresponds to stack depth and y-axis shows the cumulative percentage of dynamic context instances during execution that can be represented in a given stack depth. Thus, if a curve ascends to 100% very quickly, it means that most contexts for that benchmark could be stored with a small encoding stack, possibly even size zero. The graph for `164.gzip` is very typical for medium sized programs. Our stack is always empty while the plain stack gradually ascends. Observe the diagram of `176.gcc`. Even though the maximum encoding stack has the size of 19, our curve quickly ascends over the 90% bar with the stack depth 2. The diagrams for other large programs such as `255.vortex`, `vim` and `alpine 2.0` are similar. Finally, the diagram of `197.parser` shows that our technique is relatively less effective. The reason is that parser implements a recursive descent parsing engine [2] that makes intensive recursive calls based on the syntactic structure of input. The recursion is mostly irregular, so our simple compression does not help much. `186.crafty` is similar.

9. RELATED WORK

Explicit context encoding. The most direct approach to identifying calling contexts is to explicitly record the entire call stack. There are two main approaches for doing so. *Stack walking* involves explicitly traversing the program stack whenever a context is requested in order to construct the full identifier for the context [15]. Compared to our approach, stack walking is more expensive and hence less desirable. Maintaining the *calling context tree* [17, 19] involves explicitly unfolding a CG into a tree at runtime, with each tree path representing a context. Tree nodes are allocated and maintained on the fly. The current position in the tree is maintained by a traversal action at each call site. While call trees are very useful in profiling, our technique is more general as it provides more succinct representation of contexts

Table 1: Static program characteristics.

programs	LOC	CG nodes	CG edges	recursions	fun pointers	our max ID	BL max ID
cmp 2.8.7	6681	68	162	0	5	44	156
diff 2.8.7	15835	147	465	6	8	645	3140
sdiff 2.8.7	7428	90	281	0	5	242	684
find 4.4.0	39531	567	1362	28	33	1682	5020
locate 4.4.0	28393	320	688	3	19	251	1029
grep 2.5.4	26821	193	665	17	10	17437	44558
tar 1.16	58301	791	2697	19	46	1865752	4033519
make 3.80	29882	271	1294	61	7	551654	1543113
alpine 2.0	556283	2880	26315	302	1570	4294967295*	4.5e+18
vim 6.0	154450	2365	15822	1124	27	4291329441*	8.7e+18
164.gzip	11121	154	426	0	2	536	1283
175.vpr	29807	327	1328	0	2	1848	13047
176.gcc	340501	2255	22982	1801	128	4294938599*	9.1e+18
181.mcf	4820	93	208	2	0	6	96
186.crafty	42203	179	1533	17	0	188589	650779
197.parser	27371	381	1676	125	0	2734	14066
255.vortex	102987	980	7697	41	15	4294966803*	1.5e+13
256.bzip2	8014	133	396	0	0	131	609
300.twolf	49372	240	1386	9	0	1051	3766

*Selective reduction is applied to 288 nodes in `alpine 2.0`, 300 in `176.gcc`, 33 in `255.vortex`, and 877 in `vim`.

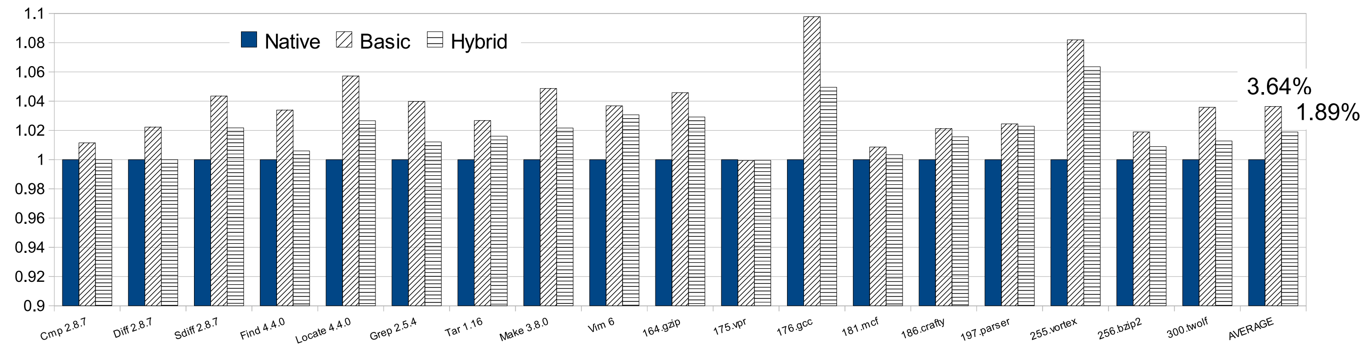


Figure 8: Normalized runtime comparison of benchmarks with no instrumentation, basic encoding instrumentation, and the hybrid encoding instrumentation. Times are normalized against the native runtime.

and has less overhead. Moreover, our encoding technique is complementary to call trees because a compressed tree can be constructed using encoding by denoting tree sub-paths as individual nodes.

Path encoding. In [4], Ball and Larus developed an algorithm to encode control flow paths. Our technique is inspired from theirs. In comparison, our algorithm is more efficient for context encoding as it encodes in a different (backward) direction. The BL algorithm breaks loop paths at control flow back edges and our algorithm similarly separates recursive contexts into acyclic subsequences and encode independently. However, due to the characteristics of contexts, our recursive encoding relies on a stack. Furthermore, we safely leverage stack offset to remove unnecessary encodings. In [12], the BL algorithm was extended to encode control flow paths across function boundaries. In [20], it was first proposed that the BL algorithm can be extended to encode calling contexts. However, the approach was not fully developed. It does not encode/decode recursive contexts. No empirical results were reported.

In [8], edge profiles are used to avoid considering cold paths and to infer paths without numbering where possible. If only a subset of paths are known to be of interest, [18] uses this information to construct minimal identifiers for the interesting paths, while allowing uninteresting paths to share identifiers when advantageous. Similar to these approaches, our technique uses profiling to guide instrumentation, improve efficiency and reduce encoding space pressure. If it were known that only a subset of calling contexts were of interest, we could further reduce both our instrumentation and identifier size, but we leave this open as future work.

Probabilistic contexts. It may be acceptable that context identifiers are not unique. That is, some arbitrary contexts may be merged with low probability. In such scenarios, *probabilistic calling contexts* can be used to very efficiently identify calling contexts with high probability of unique identifiers. In [5], calling contexts are hashed to numeric identifiers via hashes computed at each function call. Decoding is not supported in this technique. More recently, [13] uses the height of the call stack to identify calling contexts and mutates the size of stack frames to differentiate conflicting stack heights with empirically high probability. Our approach also uses the height of the call stack to disambiguate calling contexts, but in contrast, we only use it to eliminate instrumentation and path numbering where it can be shown that it is safe to do so. Their decoding is achieved by offline dictionary lookup. The dictionary is generated through training, and hence the actual calling contexts can-

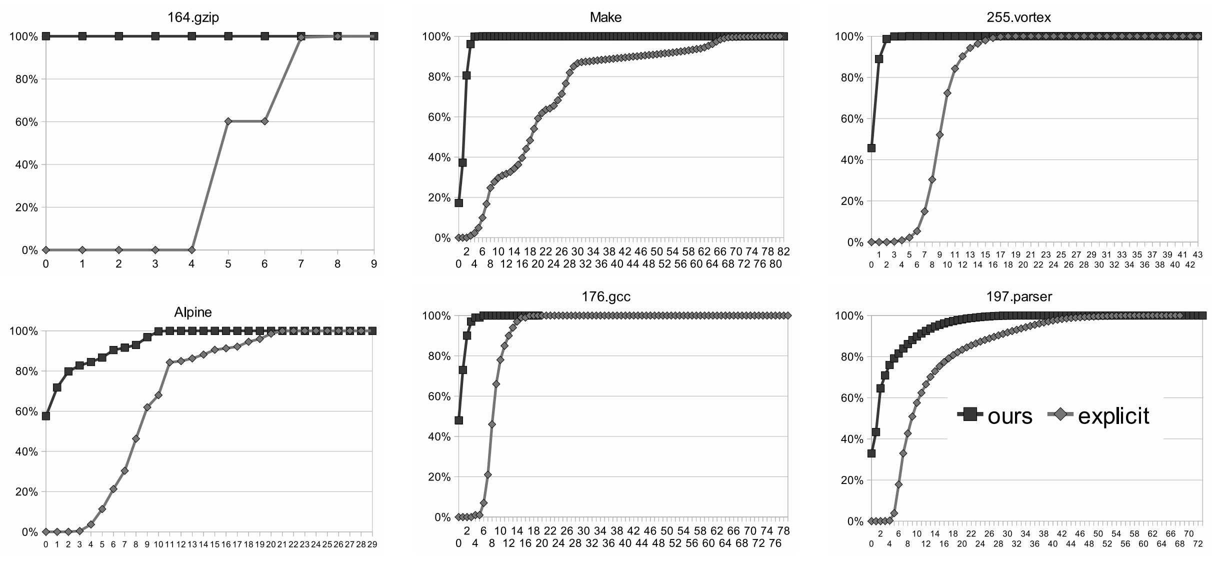


Figure 9: Context encoding stack size distributions

not always be decoded. In contrast, our approach guarantees safety and easy decoding. Furthermore, their approach is unsafe with stack allocation.

Sampling based profiling. In [23], adaptive sampling is used to guide context sensitive profiling, thus reducing the number of times contexts are even needed. While this is useful when performing hot-context profiling, it does not generalize to other uses of contexts where coverage guarantees are important.

10. CONCLUSIONS

We propose a technique that encodes the current calling context at any point during execution. It encodes an acyclic call path into a number and divides a recursive path into sub-sequences to encode them independently. It leverages stack depth to remove unnecessary encoding. The technique guarantees different contexts have different IDs and a context can be decoded from its ID. Our results show that the technique is highly efficient, with 1.89% overhead on average. It is also highly effective, encoding contexts of most medium-sized programs into just one number and those of large programs in a few numbers in most cases.

11. REFERENCES

[1] `http://www.cs.purdue.edu/~wsumner/research/cc`.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] G. Ammons, T. Ball, and J. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, 1997.

[4] T. Ball and J. Larus. Efficient path profiling. In *MICRO-29*, December 1996.

[5] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, 2007.

[6] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *ICSE*, 2009.

[7] R. Jones and C. Ryder. A study of java object demographics. In *ISMM*, 2008.

[8] R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *CGO*, 2004.

[9] Z. Lai, S. C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *FSE*, 2008.

[10] James R. Larus. Whole program paths. In *PLDI*, 1999.

[11] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, 2008.

[12] D. Melski and T. Reps. Interprocedural path profiling. In *CC*, 1999.

[13] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *OOPSLA (To appear)*, 2009.

[14] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, 2002.

[15] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[16] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *FSE*, 1997.

[17] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[18] K. Vaswani, A.V. Nori, and T.M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *POPL*, 2007.

[19] A. Villazon, W. Binder, and P. Moret. Flexible calling context reification for aspect-oriented programming. In *AOSD*, 2009.

[20] B. Wiedermann. Know your place: Selectively executing statements based on context. Technical Report TR-07-38, University of Texas at Austin, 2007.

[21] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *CGO*, 2009.

[22] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *FSE*, 2006.

[23] X. Zhuang, M. Serrano, H. Cain, and J. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI*, 2006.