8-2018

Slot-based Calling Context Encoding

Tong Zhou
University of Tennessee, tzhou9@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Tong Zhou entitled "Slot-based Calling Context Encoding." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michael R. Jantz, Major Professor

We have read this thesis and recommend its acceptance:

Micah Beck, James S. Plank

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Slot-based Calling Context Encoding

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Tong Zhou

August 2018

This thesis is dedicated to my parents.

Acknowledgments

All thanks be to my adviser, Dr. Michael Jantz. Without his support and hands-on guidance, I wouldn't have gone anywhere.

Abstract

Calling context is widely used in software engineering areas such as profiling, debugging and event logging. It can also enhance some dynamic analysis such as data race detection. To obtain the calling context at runtime, current approaches either perform expensive stack walking to recover contexts or instrument the application and dynamically encode the context into an integer. The current encoding schemes are either not fully precise, or have high instrumentation and detection overhead, and scalability issue for large and highly recursive applications.

We propose slot-based calling context encoding (SCCE), which consists of a scalable encoding for acyclic contexts and an efficient encoding for cyclic contexts. Evaluating with CPU 2006 benchmark suite, we show that our acyclic encoding is scalable, has very low instrumentation overhead, and an acceptable detection overhead. We also show that our cyclic encoding also has lower instrumentation and detection overhead than the state-of-the-art approach by significantly reducing the number of bytes pushed and checked for cyclic contexts.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Calling context enhances a wide range of software engineering fields such as debugging, and event logging [5]. For example, data race detectors record memory accesses that might cause data races. Including the calling contexts of such memory accesses will greatly help the programmers. In [17], calling context information reduces the events logged, and after removing redundant events in the replay log, the replay could be much faster. Context information is also helpful for various feed-back directed optimizations (FDO). One example is region-based memory management guided by calling contexts [8]. In our lab's researches, we link the application against a custom memory allocator which detects calling context online to guide the allocations [14]. In such FDO scenarios, if the context detection overhead is too high, it can defeat the purpose of using it as the guidance.

Over the years, many instrumentation-based encoding schemes have been proposed to efficiently encode and detect contexts. Some of them [6] [5] are imprecise approaches so different contexts can be encoded into the same value. These encodings generally have low instrumentation and detection overhead, and scale to larger applications than the precise approaches. But they are not suitable for situations where precision is required. In addition, they often provide no or complicated decodings. On the other hand, the precise approaches [13] [11] [16] usually lose efficiencies when the applications are highly recursive. It [13] has been found that when a large number of cycles are executed in the call graph, most of the instrumentation and detection overhead come from the cyclic contexts. Besides, the encoding often does not fit in one integer for large applications [11] and thus profiling is required to

prune the static call graph. The scalablity issue was addressed in [16] by extending [13]'s encoding, but in a heavyweight way which incurs complex instrumentations and greatly increases instrumentation overhead and detection overhead. Besides, the decoding also has higher complexity compared to the original decoding.

In this work, we present a precise encoding - slot-based calling context encoding (SCCE) for *large-scale highly recursive* applications. Specifically, we make the following contributions:

1. We propose an alternative encoding for acyclic contexts that scales to arbitrarily large applications, has the simplest instrumentation operation and linear-time decoding with an acceptable detection overhead.

2. We make the observation that the acyclic contexts and the cyclic contexts can be encoded independently without sharing any encoding space.

3. Based on the observation above, we present a more compact encoding scheme for cyclic contexts that significantly reduces instrumentation and detection overhead.

The rest of this thesis is organized as follows: Chapter 2 describes related work. Chapter 3 provides an overview of PCCE's acyclic encoding and SCCE's acyclic encoding. Chapter 4 provides an overview of PCCE's cyclic encoding and SCCE's cyclic encoding. Chapter 5 shows some analysis of the profile data collected from the benchmarks. Chapter 6 discusses future directions for this work, and Chapter 7 concludes the thesis.

Chapter 2

Related Work

We introduce some static and dynamic analysis works that require or benefit from context sensitivity. We then introduce some context encoding schemes.

Context sensitivity in static analysis. Context sensitivity is implemented in various static analysis, especially pointer analysis. Because time and space overhead is not a concern, these analysis typically use explicit call strings or calling context trees [9] [10] [12]. As BDD (binary decision diagram) is a widely used data structure in context sensitive pointer analysis, customized calling context numbering has been implemented to improve BDD compactness [15]. Computing this numbering, however, analyzes the entire call graph ahead of time.

Context sensitivity in dynamic analysis. Prior dynamic analyses have used either a calling context tree [3] [18] or stack walking [7] to detect context at run-time. The inefficiency of these methods are discussed below.

2.1 Non-instrumentation methods

Stack walking. Stack walking explicitly traverses the program stack whenever a context is requested. It has no instrumentation overhead because the program stack is naturally maintained, but has high detection overhead because generally programs are not directly allowed to look at its own call stack, and thus switching to the OS or the VM is required. Because of this innate security design, getting the context information from the program

execution record is prohibitively expensive and is not suitable for dynamic analysis, or FDO, etc which queries contexts very often and requires low detection overhead.

Calling context tree. Compared to static call graph, calling context tree is the dynamic execution result of a static call graph. It is widely used as a profile data, and can be dynamically maintained as the program executes. Tree may be a natural data structure to represent call paths, but it is highly uncompact. Updating the tree nodes incurs high overhead at run-time.

Last branch record (LBR). Since Haswell, the LBRs have a new mode where the CPU logs every call and return into the LBR and treats them as a stack. Typically there are 16 or 32 pairs MSRs to record a pair (from address, to address) and a top-of-stack register to indicate top of the stack. Since LBR uses architectural support and thus has very low instrumentation overhead. However, there is no efficient ways to read these MSRs at user space as of now. It has to switch into the OS code (via system calls), which has high overhead. Besides, this LBR mode is only available on certain architectures.

Function cloning. Another way to achieve context sensitivity is to duplicate a same function and let different callers call different its copies. We have observed that function cloning is a clean method that incurs almost no overhead when the cloning is only performed up to a few layers of contexts. The downside of cloning is that the code size blows up exponentially and there can potentially be an infinite amount of contexts if cycles are present in the call graph.

2.2 Instrumentation-based methods

Probablistic calling context encoding (PCC). In [6], a probabilistic calling contexts encoding is proposed. In such a probablistic encoding, same context will always be encoded into the same value, but different contexts also could be computed the same value. PCC updates an identifier with a hash function on each call site and does not treat cyclic contexts specially. They have very low probability of collision and low overhead. But it is imprecise and does not provide decoding. A more recent work [5] by the same author presents a method to reconstruct the calling context from hashed numeric identifiers. It uses the static

call graph and dynamic information to decode the calling context. To reduce the runtime overhead, it only collects dynamic information at infrequently executed callsites. However, this may cause reconstruction to fail.

Precise calling context encoding (PCCE). PCCE is a state-of-the-art precise static instrumentation-based proposed encoding method thats uniquely represent the current context of any execution point using an integer identifier and a stack. It adopts the efficient path profiling encoding approach proposed by Ball and Larus [4] and adjusts it for call path encoding. PCCE offers a nonprobabilistic approach with very low run-time overhead. However, PCCE's one drawback is that it does not scale to large applications. It relies on profiling to prune the original call graph to reduce the encoding space pressure. Besides, PCCE has an inefficient encoding for cyclic contexts. Due to these limitations, PCCE is not suitable for modern large-scale highly recursive applications.

DeltaPath. Delta builds on top of PCCE's encoding scheme, and addresses the issue of dynamic dispatch, dynamic class loading, and insufficient encoding space. The encoding space issue is resolved by introducing a set of *anchor node* and encode each anchor node's *territory* separately so that each piece is encodable with one integer. However, this solution involves complex run-time behavior such as having to push the anchor node onto a stack, which increases both instrumentation overhead and detection overhead. Besides, the decoding also has higher complexity compared to the original PCCE decoding.

Chapter 3

Acyclic Context Encoding

In this chapter, we discuss our encoding scheme for acyclic call graphs. Before introducing our scheme, we first give an overview of PCCE and use the same consistent definitions from PCCE as much as possible.

3.1 Precise Calling Context Encoding (PCCE)

For the acyclic graphs, PCCE's algorithm consists of two phases - annotation and instrumentation. During the annotation phase, each node in the call graph is assigned a value *NumCC* which represents the number of possible contexts that node can have. During the instrumentation phase, PCCE traverses all calling (incoming) edges of a given node, and increment a global identifier (*id*) by a value before the call and decrement the identifier (*id*) after the call by the same value. The value that is added and subtracted from *id* depends on the caller. The invariant is that for a given node, different taken edges produce a disjoint set value ranges of *id*. Therefore, by comparing the current *id* against these ranges, the taken edge can be uniquely determined.

Figure 3.1 shows how PCCE instruments edges on an example graph. Table 3.1 shows the context value and the number of memory operations associated with each call or return. We simulate the the memory operations incurred by the instrumentation code for the following execution path. We will run the same execution path using our encoding later.

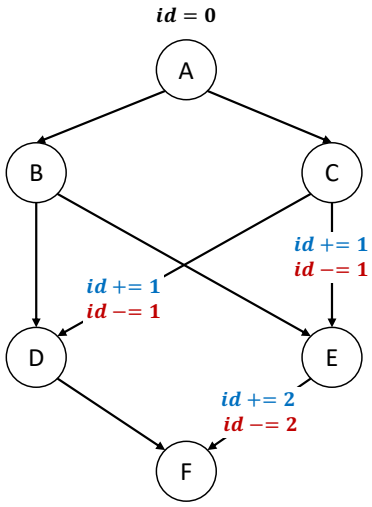

Figure 3.1: An example of PCCE's acyclic instrumentation

Table 3.1: Simulate Call Trace "ACEFECDF" With PCCE

Action	Call trace	Current context	Context ID	Total mem. ops
Call A	A	A	0	0
Call C	AC	AC	0	0
Call E	ACE	ACE	1	2
Call F	ACEF	ACEF	3	4
Return to E	ACEFE	ACE	1	6
Return to C	ACEFEC	AC	0	8
Call D	ACEFECD	ACD	1	10
Call F	ACEFECDF	ACDF	1	10

3.2 Slot-based calling context encoding (SCCE)

Similarly to PCCE, SCCE also consists of an annotation phase and an instrumentation phase. During the annotation phase, SCCE assigns a *context level* to each node in the call graph, which is defined in the next section. Besides, SCCE uses an one-byte array as the data structure to store the context as opposed to a single integer. During the instrumentation phase, code will be inserted before each instrumented edge to set one or more slots depending on the edge. No instrumentation is needed after the call. During detection, only a part of the context array is used for detection. What slots to use for detection depends on the querier function's context level.

3.2.1 Definitions

We introduce a consistent definition of the call graph with PCCE, as well as some concepts that are specific to SCCE.

Definition 3.1. *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 $\langle n, m, l \rangle$, in which $n, m \in N$, represent a caller and callee, respectively, and l represents a call site where n calls m.*

Definition 3.2. *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.*

Definition 3.3. *The context level (CL) of a node n in the call graph is the longest acyclic path between the root node and n.*

Definition 3.4. *The context level (maximum depth) of the call graph is the maximum context level of any node in the graph.*

Definition 3.5. *Context array is a global data structure used to store context. It is fixed-sized and its size equals the context level of the call graph. Each element in the array is one byte.*

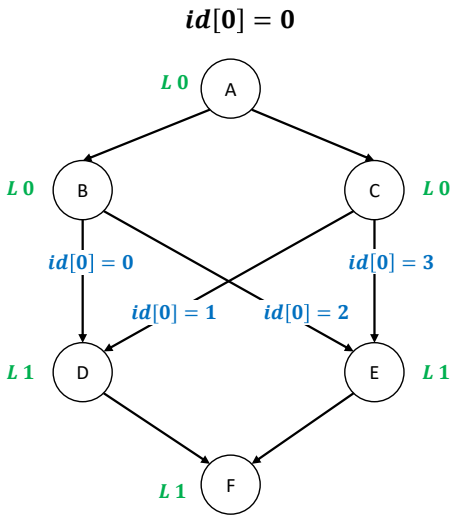

Figure 3.2: SCCE's minimal instrumentation with the same example

Table 3.2: Simulate Call Trace "ACEFECDF" With Minimal SCCE

Action	Call trace	Current context	Context array	Slots	Total mem. ops
Call A	A	A	0	-	0
Call C	AC	AC	0	-	0
Call E	ACE	ACE	3	slot 0	1
Call F	ACEF	ACEF	3	slot 0	1
Return to E	ACEFE	ACE	3	slot 0	1
Return to C	ACEFEC	AC	3	-	1
Call D	ACEFECD	ACD	1	slot 0	2
Call F	ACEFECDF	ACDF	1	slot 0	2

As a motivation example, we observe that the example in figure 3.1 can be instrumented with simpler instrumentations in figure 3.2. Table 3.2 shows the context array, the slots used to check current context ('-' means no slot is checked) and the number of memory operations associated with each call or return.

Following the motivation example, we detail SCCE's encoding and instrumentation.

3.2.2 Context Level

The context level of each node is calculated in a bottom-up fashion, we first show a naive algorithm, and then present a few special cases and optimizations that affect context levels.

Algorithm 1 Calculate Context Levels

```
1: procedure ANNOTATE(N, E)
2:     CL[root] ← 0
3:     for n ∈ N in topological order do
4:         maxCallerLevel ← 0
5:         for each incoming edge e = ⟨p, n, l⟩ of n do
6:             maxCallerLevel ← MAX(maxCallerLevel, CL[p])
7:         CL[n] ← maxCallerLevel + 1
```

- **Special Case 1.** If the node only has one caller, its context level equals its parent's level, since this subpath is unambiguous.
- **Special Case 2.** If the number of callers of node is not encodable with one slot (8 bits), the node is extended by however many levels needed to encode all its callers. For example, if a node has 300 callers, its context level would be extended by one more level, namely this node occupies two consecutive slots.
- **Optimization 1.** Semantically inlining calls can reduce the maximum depth of the call graph. In example 3.2, edge DF and EF are semantically inlined. As a result, the maximum depth of the call graph is reduced by one, and F now has 4 callers.

3.2.3 Perfect Encoding

We observe that there exists a perfect encoding for SCCE. In the most ideal case, each slot will be fully used. In this case, SCCE's representation is as compact as PCCE, but eliminates the redundancy in PCCE's instrumentation and naturally scales. Figure 3.2 is an example such a perfect encoding. While this perfectness is unrealistic in real-word call graphs, it is possible to compute an optimal SCCE encoding that is as close to the perfect encoding as possible.

3.2.4 Instrumentation

During the instrumentation, it is not always the case that the callee just has exactly one more level than the caller. There are the following two cases when there can be multiple levels' distance between the caller and the callee.

Algorithm 2 Calculate Context Levels

```
1: procedure ANNOTATE(N, E)
2:     CL[root] ← 0
3:     for n ∈ N do
4:         v = 0
5:         for each incoming edge e = ⟨p, n, l⟩ of n do
6:             ctx[CL[n]] ← v  // ctx is the context array
7:             ZEROFILL(ctx[0], ctx[CL[n] − 1])
8:             v ← v + 1
```

- **Case 1.** If the node occupies multiple levels (v can be greater than 256), v is stored to these multiple slots allocated for the node, instead of just one slot. Same goes for $v > 65536$ etc.

- **Case 2.** A forward edge exists, namely a node is found to be visited and also a descendant of current node. This gap happens because a node always takes the maximum static context level it can possible have among all paths. In such cases, unused slots are zero filled.

As a simple optimization, multiple stores on one edge can be combined as storing to a `short`, `int` or `size_t` type, depending on how many stores are combined if the combined store is aligned.

3.2.5 Decoding

Due to the fact that each call site is stored at a separate slot in the context array, decoding is simply iterating the slots up to current node's level. Note that each slot should be checked backwards down to the first slot, as the instrumentation is done bottom-up. This process does not involve any unwinding and has the complexity of $O(l)$ where l is the context level of the current node.

3.2.6 Analysis

Compared to the encoding proposed in [11] (DACCE), as an alternative scalable encoding, SCCE has much simpler instrumentation, and a linear-time decoding. Compared to PCCE,

SCCE also has simpler instrumentation, but may increase detection overhead, since it uses a less compact representation. However, our analysis on the benchmarks show that this increase in detection overhead is acceptable and can be further optimized with an ideal encoding. In addition, by not instrumenting after the call returns, the original program behavior is also retained better, since it will not interfere with some compiler optimizations such as tail call optimization.

Chapter 4

Cyclic Context Encoding

In this chapter, we discuss our encoding scheme for call graphs involving cycles.

4.1 Overview

PCCE uses the same encoding space (the integer identifier) for both back edges and non-back edges. As a result, on each back edge, a global state that represents the entire call graph execution for cyclic instrumentation is saved to the stack.

In contrast, SCCE first partitions the call graph into a cyclic and an acyclic part and encode the two parts independently. For the cyclic part, it saves a local state that represents the execution in the cyclic part for cyclic instrumentation. We first introduce our independent encoding as well as some concepts used by SCCE's cyclic encoding.

4.2 Independent Encoding

SCCE first calculates the *strongly-connected components (SCC)* of the input call graph. The SCCs naturally forms an directed acyclic graph. We define *internal edges* and *external edges* as follows:

Definition 4.1. *An internal edge is an edge that is inside a SCC in the call graph; An external edge is an edge that connects two different SCCs in the call graph.*

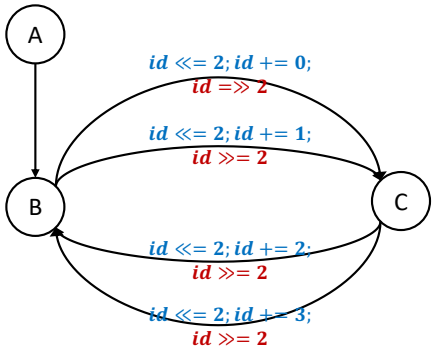

Figure 4.1: SCCE's cyclic instrumentation

Our insight is that the internal edges and external edges can be encoded independently. During the decoding, the two parts will be concatenated to form a complete call sequence. After partitioning the call graph into a cyclic part and an acyclic part, the cyclic part can be encoded separately, and potentially in a smaller space.

The idea of SCCE's cyclic encoding is also based on using slots. Instead of using one-byte slots, the cyclic encoding uses one or more bits as a slot, which yield a more compact representation but with more arithmetic operations. The insight on which we base our cyclic encoding is that the cyclic part of the graph is often much smaller than the entire graph, and thus encoding the cyclic contexts separately yields a more compact representation. SCCE uses a separate integer identifier for the cyclic contexts, and on each internal edge, some arithmetics are performed to encode this edge into the identifier. When the identifier is about to overflow, it pushes it onto the stack and reset it.

The following example gives an overview of how SCCE partitions the graph and does cyclic encoding. In this example, there are two SCCs - $\{A\}$ and $\{B, C\}$. Edge AB is an external edge, and the four edges between B and C are internal edges. The internal edges are encodable with two bits, so each slot is two bits. Instrumentation code is shown on each edge. Code in blue is inserted before the call, and code in red is inserted after the call. Before doing the bit shifting, SCCE needs to check if the identifier is about to overflow by checking if the highest slot is set. If it is set, the current identifier is pushed and reset. The guard code is omitted in the figure for the same of clarity.

4.3 Cyclic Encoding

We give more details about SCCE's cyclic encoding in this section.

For different SCCs, their internal edges are numbered in the same encoding space, instead of with regard to that particular SCC. This is because only one stack is used to save execution states from all SCCs, it makes the decoding and the implementation simpler to number all internal edges uniquely.

One optimization is that not all internal edges need to be instrumented. The entry and exit node inside a SCC can be determined via the acyclic part decoding. Given the entry and the exit, to determine the path within the SCC, not all edges in the SCC need to be instrumented. For example, if a node only has one incoming edge inside the SCC, that edge does not need to be instrumented, because the node has an unambiguous caller. This optimization slightly complicates decoding, as the acyclic contexts and the cyclic contexts cannot be decoded separately. The cyclic contexts rely on the information from the acyclic contexts, namely the entry and exit node, which makes the cyclic and acyclic part must be decoded together in the same pass.

4.4 Cyclic Decoding

Decoding is performed in a similar way as acyclic decoding. The current identifier and all integer identifiers in the stack need to be checked. Since each slot represents a taken internal edge, decoding is straightforward. When the optimization above is applied, cyclic decoding relies on information from the acyclic context. But it does not change the linear time complexity.

4.5 Hybrid Encoding

Because of our independent encoding of the acyclic and cyclic contexts, a hybrid encoding (HCCE) which uses PCCE to encode acyclic contexts, and SCCE for cyclic contexts is also possible. This hybrid encoding is desirable for situations where PCCE has a sufficient encoding space and detection happens relatively frequently. It is worth noting that

independent encoding also reduces the encoding space pressure on the acyclic part, which is separated out and has fewer edges than the entire call graph. For the largest benchmark `403.gcc` in our benchmark suite, the maximum calling context number of any node in the call graph is only about 1/22 of that of considering the entire call graph as PCCE does.

HCCE has similar decoding scheme as SCCE except the acyclic part. For the acyclic part, it'll just call PCCE's decoding routine.

Chapter 5

Evaluation

5.1 Overhead Categories

There are three categories of overheads.

- **Instrumentation overhead.** The overhead from executing the inserted code for each instrumented call site.
- **Detection overhead.** The overhead from the need to detect calling contexts at runtime. For event logging, detection involves recording the context as numeric identifiers along with an event. For memory management, detection often involves getting the context as numeric identifiers and storing it to a dictionary data structure for later lookups. Decoding is not required for detection.
- **Decoding overhead.** The overhead from decoding the numeric representation into a human readable call string.

As of now we only evaluate instrumentation and detection overhead, since decoding is usually not used online.

5.2 Platform Description

We ran all of the experiments on an Intel(R) Core(TM) i5-4590 3.3GHz CPU. The OS is RHEL server release 7.4 (Maipo). All benchmarks were compiled using LLVM 4.0 (version 4.8.5) with -O3.

5.3 Benchmarks

Our evaluation employs the standard SPEC CPU 2006 C/C++ benchmark suite [CPU]. Both profiling and performance evaluation runs use *ref* inputs. `400.perlbench`, `471.omnetpp`, `483.xalancbmk` are excluded because of tool chain errors. All benchmarks are preprocessed with the following two phases.

- **Indirect call target profiling.** We first profile the executed indirect call sites with the *ref* inputs to get a set of possible targets for each site.
- **Indirect call site promotion.** We promote the executed indirect call sites same way as how [ICP] describes.

Due to the indirect call promotion, in our evaluation, we consider the partial call graph that is reachable by the *ref* inputs. All characteristics we obtained are in terms of this partial call graph.

5.4 Implementation

Because this project is still ongoing, the current implementations for both acyclic encoding and cyclic encoding do not use the most compact representations for every benchmark. Specifically, for the acyclic encoding, we haven't implemented the optimization that reduces the maximum depth of the call graph. In other words, the encoding is not optimal. For the cyclic encoding, we are currently always using slots that are at least one byte, instead of the minimal amount of bits needed to encode internal edges. However, even so, our analysis based on the execution statistics still show a significant edge of our approach over current state-of-the-art.

Table 5.1: Call Graph Characteristics

Bench	Nodes	Total Edges	Back Edges
401.bzip2	58	2891	0
403.gcc	2326	199552	4325
429.mcf	35	580	1
433.milc	135	6656	0
444.namd	72	8350	0
445.gobmk	1632	38881	71
447.dealII	776	85186	29
450.soplex	300	18312	4
453.povray	696	66353	1026
456.hmmer	176	9373	7
458.sjeng	103	4848	7
462.libquantum	53	1123	11
464.h264ref	347	21860	42
470.lbm	28	327	0
473.astar	70	2342	0
482.sphinx3	175	9899	6

5.5 Benchmark Characteristics

Table 5.1 shows the general call graph characteristics of the benchmarks. Column *Nodes* is the number of functions in the call graph. *Total Edges* and *Back Edges* show the number of total edges and back edges respectively.

Table 5.2 shows the call graph characteristics related to PCCE. Column *Max CC Num* is the maximum number of contexts of any node in the graph. This number determines how large the acyclic identifier needs to be. *Back Edges* and *Dynamic Back Edges* show the number of static back edges and dynamically executed back edges respectively. On each dynamic back edge, PCCE pushes an identifier that is large enough to encode *Max CC Num*.

Table 5.3 shows the call graph characteristics related to SCCE. Column *Maximum Depth* is the maximum context level of any node in the graph (maximum depth of the call graph). *Internal Edges* (aka the edges inside SCCs) and *Dynamic Internal Edges* show the number of static internal edges and dynamically executed internal edges respectively. We can observe that only 4 out of 16 benchmarks have over 8 maximum depth. For depths that are less

Table 5.2: PCCE Characteristics

Bench	Max CC Num	Back Edges	Dynamic Back Edges
401.bzip2	69	0	0
403.gcc	1.84E+19	4325	2.88E+09
429.mcf	9	1	1.43E+08
433.milc	995	0	0
444.namd	84	0	0
445.gobmk	2.67E+16	71	2.35E+08
447.dealII	3525	29	1.35E+08
450.soplex	1396	4	2180
453.povray	1.74E+19	1026	7.65E+08
456.hmmer	1585	7	30
458.sjeng	32416	7	5.03E+08
462.libquantum	8621	11	0
464.h264ref	5.20E+08	42	0
470.lbm	3	0	0
473.astar	306	0	0
482.sphinx3	1562	6	2.54E+07

than 8, the entire array can be read as a 64-bit integer, thus no extra detection overhead is introduced. For depths that are over 8, the array can also be checked with strides to make detection overhead acceptable.

5.6 Pushed Bytes Analysis

The number of bytes pushed is a key factor affecting the instrumentation overhead for highly recursive applications. We use `403.gcc` as an example to show how the number of bytes is reduce with our cyclic encoding, and then show the reduction for all cyclic benchmarks. Our following comparison is between PCCE and HCCE, so this shows the effect of SCCE's cyclic encoding.

For PCCE, `403.gcc` has a maximum of 1.84E+19 calling contexts, so 8 byte is required for the PCCE id. The dynamic number of back edges is 2.88E+09. On each back edge, PCCE pushes the 8-byte id, as well as the taken edge, which is represented by one or two bytes. Therefore, the total number of pushed bytes is more than 8 * 2.88E+09 = 2.30E+10.

Table 5.3: SCCE Characteristics

Bench	Maximum Depth	Internal Edges	Dyanmic Internal Edges
401.bzip2	4	0	0
403.gcc	21	13364	5.15E+09
429.mcf	1	1	1.43E+08
433.milc	6	0	0
444.namd	2	0	0
445.gobmk	38	104	4.42E+08
447.dealII	7	29	1.35E+08
450.soplex	7	4	2180
453.povray	19	2817	1.35E+09
456.hmmer	5	8	32
458.sjeng	7	7	5.03E+08
462.libquantum	7	14	0
464.h264ref	15	87	7.62E+04
470.lbm	1	0	0
473.astar	6	0	0
482.sphinx3	7	6	2.54E+07

For HCCE, there are 13364 internal edges statically, which is encodable with 2 bytes. The dynamic number of internal edges is 5.15E+09, so the total number of pushed bytes is 2 * 5.15E+09 = 1.03E+10, and thus saving more than half of the bytes than PCCE.

Table 5.4 shows that the number of bytes pushed by PCCE divided that of HCCE for benchmarks that have cycles. This calculation assumes the most compact representation for PCCE. It shows on average, HCCE only pushes about a half of what PCCE pushes. Note that the reason why `464.h264ref` has a ratio of infinite is because the dynamic executed back edge number is 0 for PCCE, but the dynamic executed internal edge number is positive for HCCE. This is possible because there are paths that include internal edges but not back edges.

5.7 Checked Bytes Analysis

Detection overhead depends on the number of bytes that needs to be checked for detection. Table 5.5 shows that the number of bytes checked for detection by PCCE divided that of HCCE for benchmarks that check cycles during detection time. Similarly, for `464.h264ref`,

Table 5.4: Pushed Bytes (HCCE divided by PCCE)

Bench	Pushed Bytes (HCCE / PCCE)
403.gcc	0.446627959
429.mcf	1
445.gobmk	0.234686693
447.dealII	0.5000005
450.soplex	0.5
453.povray	0.441501104
456.hmmer	0.533333333
458.sjeng	0.5
464.h264ref	INF
482.sphinx3	0.5
Average	0.509165021

Table 5.5: Checked Bytes (HCCE divided by PCCE)

Bench	Checked Bytes (HCCE / PCCE)
403.gcc	0.333948
445.gobmk	0.557305
447.dealII	0.796661
453.povray	0.626218
464.h264ref	1.414741
Average	0.7457746

HCCE checked more bytes than PCCE. On average, even considering `464.h264ref`, HCCE checks about 75% of the number of bytes PCCE has to check for recursive applications.

5.8 Observations

SCCE typically reduces the number of bytes pushed and checked dynamically to 1/2 - 1/4 of PCCEs. This improvement is based on the fact that most SCCs in the call graph are small. The same technique may not work well for other graphs such as CFG (control flow graph) which typically has more back edges and internal edges.

5.9 Performance Evaluation

As stated earlier, we are still ongoing and we currently do not have the most efficient implementation of both SCCE's acyclic encoding as well as cyclic encoding. We haven't had performance results from which we can make strong conclusions at this moment.

Chapter 6

Future Work

As we are still working the project, the first priority is to finish our final implementation. Besides, we will also switch to larger-scale applications that are not encodable with PCCE. We want to measure the instrumentation and detection overhead of our acyclic encoding on these benchmarks that PCCE cannot handle directly. So far we've only shown a byte-analysis of the benefits of SCCE's cyclic encoding. To evaluate the performance benefits, we want to find highly recursive applications and compare the performance of our encoding with PCCE's.

Chapter 7

Conclusion

We first present a scalable and efficient encoding scheme for large-scale acyclic call graphs that do not fit into PCCE's encoding space. We then present an efficient encoding scheme for cyclic contexts that pushes only 50% and checks only 75% of the bytes PCCE has to push or check, by evaluating on CPU 2006 benchmark suite. We have shown great potentials of both our acyclic and cyclic encodings for large-scale highly recursive application, and we will continue the evaluation as stated in future work section.

Bibliography

[ICP] Profile-based Indirect Call Promotion. https://llvm.org/devmtg/2015-10/slides/Baev-IndirectCallPromotion.pdf. 18

[CPU] SPEC CPU 2006. https://www.spec.org/cpu2006/. 18

[3] Ammons, G., Ball, T., and Larus, J. R. (1997). Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA. ACM. 3

[4] Ball, T. and Larus, J. R. (1996). Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA. IEEE Computer Society. 5

[5] Bond, M. D., Baker, G. Z., and Guyer, S. Z. (2010). Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 13–24, New York, NY, USA. ACM. 1, 4

[6] Bond, M. D. and McKinley, K. S. (2007). Probabilistic calling context. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 97–112, New York, NY, USA. ACM. 1, 4

[7] Froyd, N., Mellor-Crummey, J., and Fowler, R. (2005). Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA. ACM. 3

[8] Jones, R. E. and Ryder, C. (2008). A study of java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 121–130, New York, NY, USA. ACM. 1

[9] Lattner, C., Lenharth, A., and Adve, V. (2007). Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA. ACM. 3

[10] Lhoták, O. and Hendren, L. (2008). Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53. 3

[11] Li, J., Wang, Z., Wu, C., Hsu, W.-C., and Xu, D. (2014). Dynamic and adaptive calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 120:120–120:131, New York, NY, USA. ACM. 1, 11

[12] Sridharan, M. and Bodík, R. (2006). Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA. ACM. 3

[13] Sumner, W. N., Zheng, Y., Weeratunge, D., and Zhang, X. (2012). Precise calling context encoding. *IEEE Trans. Softw. Eng.*, 38(5):1160–1177. 1, 2

[14] T. Chad Effler, Adam P. Howard, T. Z. M. R. J. K. A. D. and Kulkarni, P. A. (2018). On Automated Feedback-Driven Data Placement in Hybrid Memories. http://web.eecs.utk.edu/~mrjantz/papers/arcs18.pdf. 1

[15] Whaley, J. and Lam, M. S. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA. ACM. 3

[16] Zeng, Q., Rhee, J., Zhang, H., Arora, N., Jiang, G., and Liu, P. (2014). Deltapath: Precise and scalable calling context encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 109:109–109:119, New York, NY, USA. ACM. 1, 2

[17] Zhang, X., Tallam, S., and Gupta, R. (2006). Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT*

International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14, pages 81–91, New York, NY, USA. ACM. 1

[18] Zhuang, X., Serrano, M. J., Cain, H. W., and Choi, J.-D. (2006). Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 263–271, New York, NY, USA. ACM. 3

Vita

Tong Zhou is from Hubei, China. He received his Bachelor of Engineering degree from Beijing University of Posts and Telecommunications with a major in Electronic Information Engineering. Tong switched his major to Computer Science when he started his graduate study at the University of Tennessee, Knoxville. Since Januruary 2016, Tong has been working in Dr. Michael Jantz's research group.