Valence: Variable Length Calling Context Encoding

Tong Zhou
Georgia Institute of Technology
Atlanta, Georgia, USA
tz@gatech.edu

Michael R. Jantz
University of Tennessee
Knoxville, Tennessee, USA
mrjantz@utk.edu

Prasad A. Kulkarni
University of Kansas
Lawrence, Kansas, USA
kulkarni@ittc.ku.edu

Kshitij A. Doshi
Intel Corporation
Chandler, Arizona, USA
kshitij.a.doshi@intel.com

Vivek Sarkar
Georgia Institute of Technology
Atlanta, Georgia, USA
vsarkar@gatech.edu

ABSTRACT

Many applications, including program optimizations, debugging tools, and event loggers, rely on calling context to gain additional insight about how a program behaves during execution. One common strategy for determining calling contexts is to use compiler instrumentation at each function call site and return sites to encode the call paths and store them in a designated area of memory. While recent works have shown that this approach can generate precise calling context encodings with low overhead, the encodings can grow to hundreds or even thousands of bytes to encode a long call path, for some applications. Such lengthy encodings increase the costs associated with storing, detecting, and decoding call path contexts, and can limit the effectiveness of this approach for many usage scenarios.

This work introduces a new compiler-based strategy that significantly reduces the length of calling context encoding with little or no impact on instrumentation costs for many applications. Rather than update or store an entire word at each function call and return, our approach leverages static analysis and variable length instrumentation to record each piece of the calling context using only a small number of bits, in most cases. We implemented our approach as an LLVM compiler pass, and compared it directly to the state-of-the-art calling context encoding strategy (PCCE) using a standard set of C/C++ applications from SPEC CPU® 2017. Overall, our approach reduces the length of calling context encoding from 4.3 words to 1.6 words on average (> 60% reduction), thereby improving the efficiency of applications that frequently store or query calling contexts.

CCS CONCEPTS

• **Software and its engineering** → *Compilers*; *Runtime environments*;

KEYWORDS

calling context, compiler instrumentation, profiling, analysis, performance

ACM Reference Format:
Tong Zhou, Michael R. Jantz, Prasad A. Kulkarni, Kshitij A. Doshi, and Vivek Sarkar. 2019. Valence: Variable Length Calling Context Encoding. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307351

1 INTRODUCTION

Calling context (also known as *call path* or *call stack context*) is the series of active function calls that lead to a particular program location during execution. Many applications use calling context to attain better understanding of program behavior. Indeed, the ability to inspect call stack context is an essential part of a wide variety of tools for debugging[4, 9, 12, 13, 17, 18, 24, 31], testing[6, 20, 33], and analyzing [2, 11, 28, 29, 38, 41, 42] modern software.

In addition to these software diagnosis and development tools, many applications collect and use calling contexts in production systems to enhance security or help guide feedback-directed optimizations (FDOs). For instance, some frameworks monitor calling context at program system calls and raise an alert if they detect an anomalous or illegitimate call path [16, 35]. Another approach uses calling context to identify and elide redundant barriers in hot execution contexts, thereby increasing the scalability of multi-threaded applications [10]. Several works [15, 21, 23, 32, 34] improve utilization of processor caches or low-power or hybrid memory devices by building upon the idea that data objects allocated from the same calling context tend to be used for similar purposes and exhibit similar access patterns. To be successful, all of these techniques need to balance the overhead of collecting and detecting calling context with its expected benefits.

There are a number of different strategies for querying and examining calling context, each with their own advantages and limitations. A simple approach is to compute the context on demand by walking the stack upwards and recording an address associated with each stack frame (e.g., using `backtrace` [30] or `libunwind` [40]). While this approach is flexible and easy to implement, it can be very expensive, especially for applications that require precise contexts in deep call paths [32]. Another strategy is to use call path profiling to build a detailed representation of the dynamic call graph, such as a calling context tree [3]. While these structures are very useful for understanding dynamic program behavior, storing and updating them is often too expensive for online applications. Some researchers have developed adaptive bursting and/or sampling-based strategies to curtail these overheads, but these techniques are imprecise and can still incur high costs during the sampling period [14, 44].

An option with lower overhead is to use compiler instrumentation at each function call/return to track and encode the calling context in a small area of memory. Bond and McKinley used this approach to maintain the current calling context in a probabilistically unique hash value stored as a single word [8]. While this strategy has low instrumentation overhead and is very compact, the contexts it produces are not always distinct (i.e., there may be collisions on the hash) and are not decodable. Bond et al. later extended this approach to support call path decoding, but this extension requires additional instrumentation for accurate decodings [7]. Each decoding operation also entails an expensive search of a large space of context values, and is only suitable for offline application.

Some other encoding strategies sacrifice compactness for accurate and reliable decoding. Sumner et al. adapted the Ball-Larus path profiling technique [5] to implement precise calling context encoding (PCCE) [37]. Their approach employs static call graph analysis to construct instrumentation that always generates unique encodings for every possible calling context. This approach can encode all of the non-recursive call path contexts using only a single 4- or 8-byte word for most small and medium-sized applications. To record recursive contexts, it pushes the current encoding onto a runtime stack each time the application makes a recursive call. While this approach works well in many cases, scaling it to larger applications is not trivial, and it often generates lengthy encodings for highly recursive call paths. More recently, some other works have extended PCCE with more flexible encoding mechanisms for applications with larger call graphs as well as dynamic and indirect calls [27, 43]. However, these scalable approaches generate lengthier encodings than PCCE, which can slow down context detection and decoding.

This work presents *Valence* (***Va**riable **len**gth calling **c**ontext **e**ncoding*), a novel instrumentation-based approach for efficient encoding of program calling contexts. Valence generates precise and scalable calling context encodings that are, on average, only 37% of the size of encodings produced by prior techniques. The key insight behind our approach is that it is often not necessary to update or store an entire word to encode each part of the calling context. Our approach uses static analysis of the relative position of nodes in the call graph to annotate each edge with a small number of bits that are sufficient to encode each potential calling context. Then, rather than sum or hash these values into the same word or set of words, the instrumentation appends them to a list of bits whenever the application makes a function call, thereby producing a variable length encoding.

Compared to current precise encoding techniques, our approach requires similar storage to record acyclic (i.e., non-recursive) call paths. However, we find that most long call paths are actually composed of a series of relatively small recursive cycles. While current techniques store the entire current context value for each recursive cycle, Valence represents the entire cycle in only a few bits, in most cases, allowing it to reduce the size of the context encoding significantly. Furthermore, Valence scales naturally to applications with larger call graphs and enables a more efficient decoding scheme than prior approaches.

This work makes the following important contributions:

(1) We present Valence, a novel calling context encoding approach that uses variable length instrumentation to generate precise and scalable encodings that are more compact and decode more efficiently than previous techniques.
(2) We develop an LLVM [26] compiler pass that automatically applies Valence instrumentation to application binaries. Instrumented applications interface with a lightweight runtime system that intercepts and queries the calling context during execution.
(3) We evaluate Valence using C and C++ applications from the standard SPEC CPU® 2017 benchmark suite. Our experiments show that Valence reduces the length of calling context encoding by almost 90% in the best case, and over 60% (from 4.3 to 1.6 64-bit words), on average, with little or no impact on instrumentation costs.

2 BACKGROUND

2.1 Precise Calling Context Encoding

For baseline comparison to our approach, we implement and employ the current state-of-the-art technique for precise calling context encoding (PCCE) [37].

Definitions. The call graph (CG) is formally defined as a pair $\langle N, E\rangle$, where N is a set of nodes with each node representing a single function, and E is a set of directed edges. Each edge $e \in E$ is a triple $\langle p, n, l\rangle$, where $p, n \in N$ are a caller and callee nodes, and l is a call site where p calls n. Indirect calls and virtual calls handling is discussed in 4.

A calling context (CC) of an invocation of function f is defined as a path in the CG from the root leading to the node representing f. During execution, PCCE maintains the current calling context in a value known as *id*.

Encoding Acyclic Contexts. The key concept of PCCE is that, at a given node n, it is possible to compute the total number of non-recursive calling contexts, denoted as $numCC(n)$, statically. Therefore, each of the calling contexts at node n can always be uniquely encoded by a number in the range $[0, numCC(n))$. To generate its instrumentation, PCCE first annotates each node n with $numCC(n)$, such that $numCC(n) = \sum_{i=1}^{m} numCC(p_i)$ where p_i are the m parents of n. Next, it iterates over the incoming edges of each node in topological order. For each edge $e = \langle p, n, l\rangle$, it computes a sum $En(e)$, where $En(e)$ is zero if e is the first incoming edge to node n, and otherwise, $En(e) = \sum numCC(p_j)$, where p_j are the parents of the previous incoming edges to n. PCCE then inserts instrumentation on e as follows: immediately before the call site l, add $En(e)$ to *id* to record the call, and immediately after l, subtract $En(e)$ from *id* to restore *id* to its original value.

Consider the example in Figure 1, which is taken from [37]. The boxes next to each node show their $numCC(n)$ values, while the edge labels show the actual instrumentation for each call site. For instance, since I's incoming edges, FI and GI, precede JI in a topological traversal, the instrumentation value for call site JI =

call path	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
AJG	2
AJGI	6
…	

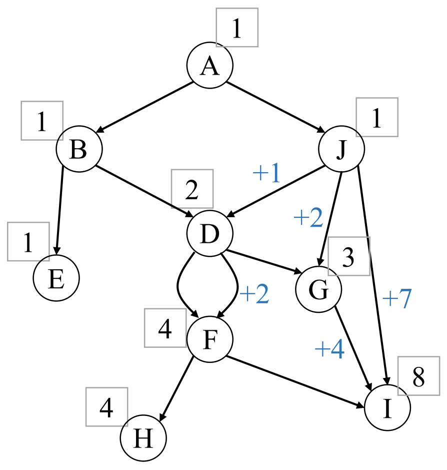


Figure 1: Encoding non-recursive calling context with PCCE. The boxed number next to each node is *numCC*(*n*). The edge label "+c" means "id += c" is added before the call site, and "id -= c" immediately after. The superscripts on *D* distinguish the two call sites from *D* to *F*.

$numCC(F) + numCC(G) = 3 + 4 = 7$. Thus, PCCE records a call from *J* to *I* by adding 7 to *id*. For many call sites, such as *AB*, *BD*, and *DF*, it is not necessary to insert instrumentation because PCCE does not update *id* (i.e., $En(e) = 0$) for exactly one incoming edge of each node. Note that, while this approach does not generate a different *id* for every program path, it does ensure that *id* uniquely encodes the distinct calling contexts at each program node, as shown in the table on the left in Figure 1.

Recursive Calling Contexts. For programs that use recursion, PCCE encodes the acyclic sub-paths of each recursive cycle in a runtime stack. When the application makes a recursive call, it pushes the current acyclic context *id* as well as an identifier for the recursive call site onto the stack, and then records the following acyclic context in a new *id* value. To implement this approach, PCCE inserts a second "dummy" root node in the call graph with dummy outgoing edges to: a) the original root node, and b) any node that is a target of a backward edge (i.e., recursive call) in the original call graph. It then annotates the modified graph as described above, instruments the (non-dummy) forward edges to encode the acyclic contexts, and inserts code for each backward edge to push and pop the current acyclic context before and after the recursive call site.

Decoding PCCE Context. To decode context values, PCCE traverses the call graph upwards (i.e., towards `main`) from a given node *m*. At each step, subranges of $[0, numCC(m))$ uniquely identify the next piece of the calling context. Specifically, decoding proceeds by determining the subrange $[En(e_i), En(e_{i+1}))$ to which the current *id* belongs, where e_i are the incoming edges to the query point, visited in topological order. At each step, the decoder sets *id* to $id - En(e_i)$, where e_i is the edge corresponding to the correct subrange. The algorithm terminates when it reaches the root of the call graph. Consider the example of decoding the context from node *I* in Figure 1. The context values $0-3$ indicate that *I* must have been called from *FI*, while values $4-6$ correspond to a call from *GI*, and 7 to a call from *JI*.

Recursive context decoding is straightforward: the algorithm simply processes one acyclic sub-path of the recursive context at a time, until all values on the stack are fully decoded. Additional and more specific details about the decoding process are available in [37].

2.2 PCCE Limitations and Drawbacks

While PCCE is typically much more efficient than naïve strategies for tracking and querying the calling context, it still has some limitations, including: 1) it is difficult to apply PCCE if the combinatorial explosion of paths is so large that representing them exceeds the range of a single word on the target machine (which is often the case for many large and object-oriented applications, even on modern 64-bit architectures [43]), 2) it only encodes calling contexts for static call graphs, and lacks facilities for dynamic libraries and indirect calls, and 3) recursive calling contexts can grow to include dozens, hundreds, or even thousands of words (see Figure 4 in Section 6.1). Such lengthy encodings have high detection costs and add significant pressure to processor caches.

Some later efforts partially addressed these limitations, but their underlying encoding schemes still use the original PCCE technique. For instance, Zeng et al. [43] developed an extension to PCCE that flexibly supports larger and more complex call graphs. Their approach, called DeltaPath, divides the call graph into separate *territories*, and uses PCCE to encode calling context from different territories as distinct values on a runtime stack. While this technique avoids the potential issue of integer overflow in the context value, it requires additional storage to encode each territory. Indeed, a straightforward implementation that uses separate words for the territory and context values will generate encodings that are at least twice as large as those of PCCE alone. Li et al. [27] proposed to integrate PCCE with DACCE, a lightweight dynamic binary instrumentation (DBI) runtime system that discovers the call graph during execution. Their approach inserts PCCE instrumentation dynamically and re-encodes the calling context when necessary, allowing it to support dynamic and indirect function calls automatically.

Since our work focuses on the potential benefits of an alternative context encoding scheme, we compare our approach directly to the original PCCE technique using only static instrumentation. Our encoding scheme scales naturally to large and object-oriented call graphs, and is often much more compact than PCCE and DeltaPath, especially for recursive contexts. While our current implementation only supports static call graphs, its benefits could be extended to applications with dynamic libraries and indirect function calls by integrating it with the DACCE runtime.

3 VARIABLE LENGTH CALLING CONTEXT ENCODING

We now present our approach for encoding application calling context with variable length instrumentation, which we call Valence. As in past work, Valence uses different strategies for recursive and non-recursive calling contexts, which we describe in separate subsections below. The following descriptions use definitions for the call graph and calling context that are consistent with those listed in Section 2.1.

3.1 Variable Length Instrumentation for Acyclic Calling Context

Valence records the calling context as a list of variable-length bit values, where each value on the list represents a single edge in the call path. Previous techniques that use a sum or hash function to encode the calling context can generate values that exceed the range of a 32- or 64-bit integer for many real-world programs. In these cases, performance degrades significantly since most modern machines do not support native addition and subtraction operations on such large integer types. In contrast, our approach scales naturally on any architecture that can manipulate a list of bits with shift and logical operations.

To avoid lengthy calling contexts, Valence aims to use as few bits as possible to record each step in the call path. Consider that, given a function f with c potential callers, a value with only $lg(c)$ bits is sufficient to distinguish the immediate parents of f. For static call graphs, the value c is easily computed as the number of incoming edges to the node corresponding to f. Thus, a simple approach would be to enumerate the incoming edges of each node and assign each edge a unique value between 0 and c. Then, at each function call (and return), the application could append (remove) $lg(c)$ bits containing the corresponding edge value to the end of a list that represents the current calling context. Unfortunately, this approach must also maintain a pointer to the end of the context list, which increases its instrumentation costs.

Valence addresses this drawback for *acyclic* call graphs using static analysis and annotation. Specifically, Valence annotates each node with a level value. The *level* of each node refers to the maximum number of bits necessary to encode its potential calling contexts. Level annotations allow the encoding scheme to insert information about each (non-recursive) function call into a specific range of bits, rather than append it to a list. Moreover, since the length and position of each piece of the encoding is known statically, this approach is able to label many of the call graph edges with a default value of zero and omit their instrumentation entirely.

Algorithm 1 presents pseudocode for encoding acyclic calling context with Valence. Valence applies the function call instrumentation in two separate passes over the call graph: the first pass, shown in the `annotate` procedure, computes the level of each node and creates bit codes for each edge, while the second pass, in the `instrument` procedure, inserts instructions to encode the calling context at each call site.

Each node may have incoming edges that originate at different levels, as well as multiple incoming edges that originate from the same level. To encode each function call, our approach uses a two-layer encoding where: 1) the *level code* indicates the level of the

Algorithm 1 Encoding for Acyclic Call Graphs

```
1:  procedure ANNOTATE($N, E$)
2:      for each node $n \in N$ in topological order do
3:          $levelCount \leftarrow 0$
4:          $edgeCounts \leftarrow$ Map⟨Integer, Integer⟩()
5:          $n.lvCodes \leftarrow$ Map⟨Integer, BitList⟩()
6:          $n.c2lv \leftarrow$ Map⟨BitList, Integer⟩()
7:          $n.ccMap \leftarrow$ Map⟨BitList, Map⟨BitList, Edge⟩⟩()
8:
9:          for each incoming edge $e = \langle p, n, l \rangle$ of $n$ do
10:             if $p.level \notin n.lvCodes$ then:
11:                 $n.lvCodes[p.level] \leftarrow$ bits($levelCount$)
12:                 $n.c2lv[n.lvCodes[p.level]] \leftarrow p.level$
13:                 $edgeCounts[p.level] \leftarrow 0$
14:                 $ecodes \leftarrow$ Map⟨BitList, Edge⟩()
15:                 $n.ccMap[n.lvCodes[p.level]] \leftarrow ecodes$
16:                 increment $levelCount$ by 1
17:
18:             $e.code \leftarrow$bits($edgeCounts[p.level]$)
19:             increment $edgeCounts[p.level]$ by 1
20:             $n.ccMap[n.lvCodes[p.level]][e.code] \leftarrow e$
21:
22:         $n.maxLen \leftarrow 0$
23:         for each $\langle level, numEdges \rangle \in edgeCounts$ do
24:             $codeLen \leftarrow level$+length(bits($numEdges$))
25:             $n.maxLen \leftarrow$max($codeLen, n.maxLen$)
26:
27:         $n.lvCodeLen \leftarrow$ length(bits($n.lvCodes$.size()))
28:         $n.level \leftarrow n.maxLen + n.lvCodeLen$
29:
30: procedure INSTRUMENT($N, E$)
31:     for each edge $e = \langle p, n, l \rangle \in E$ do
32:         $cc \leftarrow e.code \cdot n.lvCodes[p.level]$
33:         if $cc$ has any nonzero bits then
34:             set $id[p.level, n.level)$ to $cc$ before site $l$
35:             reset $id[p.level, n.level)$ after site $l$
```

caller function, and 2) the *edge code* distinguishes incoming edges from the same level. Lines 9-20 of the `annotate` procedure generate these codes for each edge. For each node n in the call graph, the algorithm builds a structure ($n.lvCodes$) that maps the levels of n's parent nodes ($p.level$) to distinct level codes. Similarly, for each incoming edge to n, it generates an edge code ($e.code$) that distinguishes the edge from other edges with parent nodes of the same level. To produce unique level and edge codes, the algorithm maintains integer counts (i.e., $levelCount$ and $edgeCounts[p.level]$) and converts the counts into distinct bit codes using the `bits` procedure.[1] The remainder of the `annotate` procedure (lines 22-29) uses these codes and their values to calculate the maximum length of the calling context encoding at node n, which is stored on the node as $n.level$. Note that the other structures shown in `annotate`, including $n.c2lv$, which maps a given level code to a specific level value, and $n.ccMap$, which maps a pair of level and edge codes to a

[1]Specifically, `bits` returns a compact list of bits encoding the integer value passed as its argument.

call path	id
AB	-
AJ	-
ABD	0
AJD	1
ABE	-
ABD^1F	00
AJD^1F	10
ABD^2F	01
AJD^2F	11
AJG	01
AJGI	0110
…	

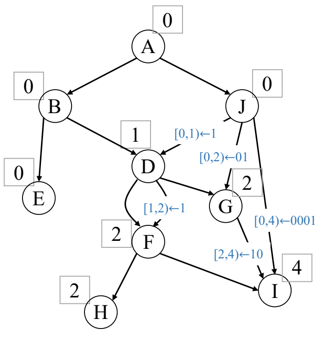
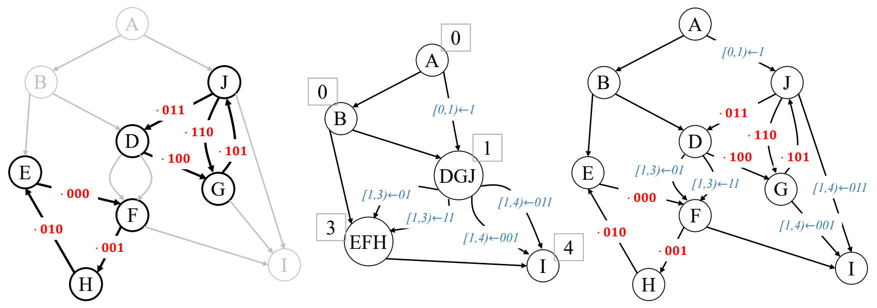


Figure 2: Encoding acyclic calling context with Valence. The boxed numbers show the level of each node. The edge label "$[n, m) \leftarrow v$" means "set bits n through $m-1$ in *id* to v" is added immediately before the call site, and "reset bits n through $m-1$ in *id*" immediately after.



(a) Complex SCCs with cyclic instrumentation

(b) Transformed call graph with acyclic instrumentation

(c) Original call graph with combined instrumentation

Figure 3: Encoding recursive calling context with Valence. The edge label "$\cdot v$" means "push v onto *recID*" is added immediately before the call site, and "pop v from *recID*" immediately after. *Italic blue* edge labels update *acycID*, while bold red edge labels update *recID*.

specific edge in the call graph, are useful for decoding, but are not necessary for encoding calling contexts.

After annotation, the `instrument` procedure traverses the edges in the call graph a second time to apply the Valence instrumentation at each call site. Specifically, for each incoming edge of each node, it concatenates the appropriate edge and level codes to create a combined code *cc*. It then uses the level annotations of its source and destination nodes to insert instrumentation that records *cc* into the appropriate range of bits in the calling context *id*.

Consider again the example call graph from [37], now shown in Figure 2. In this figure, the boxed numbers show the level of each node, while the edge labels show the Valence instrumentation, including the range of bits within *id* and the bit codes that are used at each call site. The table on the left shows the calling context encoding (as a list of bits) after traversing some example call paths. Observe that the parents of node I have two distinct levels, 2 and 0, which are assigned level codes 0 and 1, respectively. Since edges FI and GI both originate from parents at level 2, Valence further distinguishes them with different edge codes, resulting in combined codes of '00' and '10' for these edges, respectively.[2]

Similar to previous techniques, Valence is able to omit instrumentation for exactly one incoming edge of each node, when the level and edge codes do not contain any nonzero bits. In some cases, the encoding for an incoming edge may require fewer bits than the levels of the source and destination nodes allow. For instance, only one bit (just the level code) is necessary to distinguish edge JI, but the encoding scheme reserves four bits for its call site. In these cases, the instrumentation pads the context codes with leading zeros to ensure the encoding is available at the correct position.

Decoding Acyclic Calling Context. Algorithm 2 presents the approach for decoding non-recursive calling contexts with Valence. The decoding process starts from a query point in a given node m,

[2]The level code is always placed after the edge code (on the right, in this example) to facilitate decoding.

Algorithm 2 Decoding for Acyclic Call Graphs

```
 1: procedure DECODE($id, m$)
 2:     $cc \leftarrow$ "$m$"
 3:     while node $m \neq root$ do
 4:         $lvHigh \leftarrow m.level$
 5:         $lvLow \leftarrow (lvHigh - m.lvCodeLen)$
 6:         if $m.lvCodeLen > 0$ then
 7:             $lvCode \leftarrow id[lvLow, lvHigh)$
 8:
 9:         $edgeHigh \leftarrow lvLow$
10:         $edgeLow \leftarrow m.c2lv[lvCode]$
11:         if $edgeCodeLen > 0$ then
12:             $edgeCode \leftarrow id[edgeLow, edgeHigh)$
13:
14:         $\langle p, n, l \rangle \leftarrow m.ccMap[lvCode][edgeCode]$
15:         $cc \leftarrow$ "$p^l$" $\cdot\ cc$
16:         $m \leftarrow p$
17:     return $cc$
```

and proceeds upwards towards the root function. At each step, it first decodes the level of the next function in the calling context using call graph annotations to determine the range of indexes that contain the current level code. Given the level code, it can then determine the position and value of the edge code, and lookup the actual edge corresponding to the codes using the *ccMap* structure on the current node m.

While decoding efficiency is not a primary concern for many applications, it is important to note that this approach does have some performance advantages compared to prior techniques. For instance, each step in the PCCE decoding process compares the context value to a set of ranges corresponding to the incoming edges of the current node. In contrast, Valence uses more detailed call graph annotations, including the *ccMap* structure on each node,

Algorithm 3 Encoding for Recursive Call Graphs

```
1:  procedure ANNOTATEREC($N, E$)
2:      labelSCCs($N, E$)
3:      labelCyclicEdges($N, E$)
4:
5:      $curEdgeCode \leftarrow 0$
6:      for each cyclic edge $e \in E$ do
7:          $e.code \leftarrow$bits($curEdgeCode$)
8:          $recMap[e.code] \leftarrow e$
9:          increment $curEdgeCode$ by 1
10:     $recCodeLen \leftarrow$length(bits($curEdgeCode$))
11:
12:     $(N', E') \leftarrow$ getAcyclicGraph($N, E$)
13:     annotateAcyc($N', E'$)
14:
15: procedure INSTRUMENTREC($N, E$)
16:     for each edge $e = \langle p, n, l \rangle \in E$ do
17:         if $e$ is a cyclic edge then
18:             push $e.code$ onto $recID$ before site $l$
19:             pop $e.code$ from $recID$ after site $l$
20:         else
21:             instrumentAcyc($e, acycID$)
```

to determine the next edge in the calling context in constant time. (Of course, the size of the static annotations grows with the number of nodes and edges in the call graph.) Moreover, this approach allows some other types of queries, such as whether a particular edge was taken or not on the path to the current node, to complete in constant time.

3.2 Encoding Recursive Calling Contexts

Static length structures are not sufficient for recursive calling contexts, which may include an arbitrary number of cycles in the call path. To encode cyclic contexts, Valence distinguishes edges that might be part of a recursive cycle by identifying and annotating the strongly connected components in the call graph. This approach allows Valence to encode potentially cyclic contexts into an *unbounded* stack structure, separate from the static length array that is used for acyclic call paths.

Formally, a *strongly connected component* (SCC) of the graph $G = (N, E)$ is a subgraph (N', E') where every node $n \in N'$ is reachable from every other node in N'.[3] Observe that any edge in E that begins and ends in the same SCC must be part of a cycle. We refer to such edges as *cyclic edges*, and call any SCC that contains at least one cyclic edge a *complex SCC*.

Algorithm 3 presents pseudocode for instrumenting recursive applications with Valence. Similar to the acyclic encoding, the algorithm applies the instrumentation in two phases, shown in the `annotateRec` and `instrumentRec` procedures, respectively. First, `annotateRec` labels the SCCs in the original call graph[4], and marks the edges that begin and end within the same SCC as cyclic edges. Next, in lines 5-10, it assigns unique context codes to each cyclic edge, and builds the *recMap* structure to map each code back to its corresponding edge during decoding. Note that all cyclic edges receive distinct context codes, even if they belong to different SCCs. While this approach requires lengthier encodings than the alternative, it also allows Valence to distinguish edges from different SCCs, which is important for correct decoding. The remainder of `annotateRec` uses Algorithm 1 from the previous subsection to annotate the remaining (non-cyclic) edges in the call graph.

[3]By this definition, an acyclic graph $G = (N, E)$ contains exactly N SCCs.
[4]Our current implementation of `labelSCCs` uses Tarjan's algorithm [39].

Algorithm 4 Decoding for Recursive Call Graphs

```
1:  procedure DECODEREC($acycID, recID, recTop, m$)
2:      $cc \leftarrow$ "$m$"
3:      while node $m \neq root$ do
4:          if $recID$ is not empty then
5:              $recLow \leftarrow recTop - recCodeLen$
6:              $code \leftarrow recID[recLow, recTop)$
7:              $\langle p, n, l \rangle \leftarrow m.recMap[code]$
8:              if $n = m$ then
9:                  $recTop \leftarrow recLow$
10:             else
11:                 $\langle p, n, l \rangle \leftarrow$ decodeAcyc($acycID, m$)
12:         else
13:             $\langle p, n, l \rangle \leftarrow$ decodeAcyc($acycID, m$)
14:         $cc \leftarrow$ "$p^l$" $\cdot$ $cc$
15:         $m \leftarrow p$
16:     return $cc$
```

The `instrumentRec` procedure is mostly similar to the approach for instrumenting acyclic call graphs, but includes an additional case to append the codes for each cyclic edge to an unbounded stack (*recID*) rather than a static length array (*acycID*). The call to `instrumentAcyc` corresponds to lines 32 - 35 in Algorithm 1, where the first and second arguments are the instrumented edge e and context value *id*, respectively.

Consider the recursive call graph in Figure 3, which contains two complex SCCs, *EFH* and *DGJ*, with a total of seven cyclic edges. In this example, Valence pushes only three ($ceil(lg(7))$) bits to the *recID* on each function call corresponding to each cyclic edge. To instrument the non-cyclic edges in this call graph, we apply the acyclic instrumentation procedure (Algorithm 1) to the graph that results from consolidating each complex SCC into a single node, as shown in Figures 3(b) and 3(c).

Decoding Recursive Calling Contexts. Algorithm 4 presents the approach for decoding calling contexts that might include recursive call paths. Similar to the procedure for acyclic calling contexts, the algorithm decodes the context one edge at a time, and uses call graph annotations to determine the number of bits to read from the context values. To decide whether to use the recursive context stack *recID* or the acyclic value *acycID*, each step considers whether *recID* is non-empty, and if so, whether the top edge in the stack leads to the current node *m*. If both conditions are met, it decodes the next piece of the context from *recID*, and otherwise, from *acycID*. The calls to `decodeAcyc` in Algorithm 4 correspond to the inner part of the while loop (lines 4 – 15) in Algorithm 2.

Intuitively, the decoder always unwinds the *recID*, if possible, and proceeds using the *acycID* otherwise. In most cases, only one of either *recID* or *acycID* will have a leading edge that ends at the current node, but there are scenarios where both context values appear to be valid options. For instance, consider when the application corresponding to Figure 3 traverses the path AJD^1FHEF, and attempts to decode the calling context at node F. In this situation, the code for the edge EF would be the top item of *recID*, while the code for D^1F would be the last part of *acycID*. In such cases, unwinding the *recID* first is always correct because the alternative option of following the acyclic context will never reach the current node again (if it did, it would be encoded as cyclic context), and there would never be an opportunity to decode the *recID*.

3.3 Hybrid Calling Context Encoding

While the acyclic encoding strategy of Valence is more scalable and enables more efficient decoding than PCCE's acyclic approach, it also produces lengthier encodings in some cases. For instance, in Figure 2, Valence uses four bits to encode the calling contexts at node I while PCCE generates a maximum *id* of seven for the same call graph (as shown in Figure 1), which could allow it to encode all of the potential calling contexts at node I in only three bits. To better understand the limits and potential of our approach, we also developed a hybrid calling context encoding (HCCE) that uses PCCE for the acyclic parts of the call graph and Valence for the cyclic edges of each strongly connected component. In this way, HCCE sacrifices scalability and decoding efficiency for potentially more compact encodings than the original Valence.

4 IMPLEMENTATION DETAILS

We implemented each encoding strategy using a custom LLVM [26] compiler pass as well as a small set of library routines written in C++. To prepare each instrumented executable file, we first compile, optimize, and link the application's source code into a single call graph composed of LLVM IR. Next, we employ a profile-driven compiler pass [22] to remove the virtual and indirect calls from the graph and replace them with a set of direct calls. We then apply one of our custom passes to annotate the resulting call graph with data and instructions that implement one of the encoding algorithms described above. For example, the PCCE pass inserts three instructions (e.g., load, add, store) on edges that add or subtract an integer from the current *id*. Likewise, the Valence pass inserts bit manipulation instructions on edges that set or reset bits in the acyclic calling context.

To simplify the instrumentation of recursive call paths, our implementation uses an external library routine to insert and remove context values from an unbounded stack structure. Specifically, the PCCE instrumentation invokes the external library on each back edge to push or pop a word from the context stack. Since the Valence strategy adds only a few bits at a time to the *recID*, it is not necessary to invoke an external routine on every cyclic edge. Instead, our implementation represents the *recID* as a stack of words, and maintains a count of the number of bits that have already been added to the top word of the stack. When the application traverses a cyclic edge, the instrumentation checks if appending the edge code to the top word of *recID* will cause an overflow, and if so, it invokes the library to push a new word onto the stack. Otherwise, it simply shifts the bits in the top word of the *recID* and uses bitwise operations to insert the code.

It is important to note that the implementation of Valence involves engineering choices and tradeoffs that can impact its efficiency. For instance, our current instrumentation does not attempt to split edge codes across word boundaries, and thus, may leave some bits unused at the end of some words. Another example is that, in many cases, the acyclic calling context encoding will not align with word boundaries, which will result in unused bits in the last word of the *acycID*. In these cases, our implementation does attempt to use this space to encode the first few edges of the *recID*. Such choices can impact the compactness and instrumentation overhead of Valence, especially for smaller applications.

5 EXPERIMENTAL SETUP

5.1 Benchmark Description

We evaluate each encoding approach using 14 applications from the SPEC CPU® 2017 [36] benchmark set. Although our LLVM-based implementation supports the use of Fortran (through tools such as the `Flang` [1] front end), our evaluation focuses on C/C++ applications, which tend to use recursion more frequently and have larger and more complex call graphs. Specifically, our selected benchmarks include all of the C and C++ applications from SPEC CPU® 2017, with the following exceptions: 1) we exclude *perlbench* and *blender* due to their use of the `setjmp`/`longjmp` facilities for exception handling[5] 2) we exclude *omnetpp* because applying the adopted indirect call promotion pass [22], described in Section 4, fails due to a segmentation fault. Additionally, the acyclic call graph for *gcc* contains too many calling contexts to encode with either PCCE or HCCE on our 64-bit platform. For comparison purposes, we chose to include results for PCCE and HCCE and simply allow overflow in the acyclic calling context. Thus, the instrumentation and detection results for PCCE and HCCE for *gcc* only provide a conservative estimate of the actual costs of these approaches.

The first six columns in Table 1 list each of our selected benchmarks along with information regarding the size and structure of their static call graphs. Thus, the selected applications exhibit very different call graph sizes and complexities. The smallest benchmark includes less than 20 nodes, while the largest contains many thousands of nodes and edges. Overall, 12 of the 14 benchmarks have at least one recursive cycle, and for some benchmarks, such as *gcc* and *povray*, a significant portion of the call graph edges are part of a cycle.

5.2 Experimental Platform and Configuration

All of our experiments were run on a Dell OptiPlex 7020 desktop machine with a 3.30GHz Intel® Core™ i5-4590 CPU. This machine supports the standard `x86_64` instruction set and has a 64-bit word size. Its processor includes four compute cores with a shared 6MB L3 cache, and is connected to 16GB of DDR3 1600MHz DRAM. We

[5]PCCE supports `setjmp` and `longjmp` by saving a copy of the height of the context stack and the current context identifier within the local memory of the function that is active whenever `setjmp` is encountered. It then uses this information to restore the context when the application returns with `longjmp`. We plan to support these facilities with Valence using a similar mechanism, but have not yet completed this part of our implementation.

Table 1: Benchmark statistics. From left to right, the columns show: benchmark name, # of CG nodes, # of CG edges, # of (simple and complex) SCC's, # of SCCs with at least one cyclic edge, # of edges that are part of a cycle, # of bits needed to encode acyclic call graphs with each encoding approach (PCCE, Valence, and HCCE), average and maximum length of the calling context without encoding (one word per edge), # of function calls made during the run (K=thousand, M=million, B=billion).

Benchmark	Call Graph Statistics					Acyclic Encoding Bits			Runtime Statistics		
	Nodes	Edges	SCCs	Complex SCCs	Cyclic Edges	PCCE	Valence	HCCE	Avg. CC Length	Max. CC Length	Func. Calls
gcc	19,011	131,388	17,182	459	28,330	214	148	66	23.11	2,581	54.2 B
mcf	32	126	32	1	1	6	7	6	8.39	29	20.1 B
cactuBSSN	1,048	22,820	1,040	12	67	28	55	29	16.96	61	8.75 M
namd	61	553	61	0	0	10	13	10	3.99	4	559 M
parest	1,315	15,080	1,234	46	200	25	42	25	25.60	49	585 B
povray	519	8,258	377	57	2,380	57	57	36	15.07	41	50.0 B
lbm	17	27	17	0	0	2	2	2	1.00	2	6.06 K
xalancbmk	4,055	22,848	3,700	181	1,566	42	87	40	16.69	83	18.1 B
x264	367	2,318	366	1	2	18	27	18	5.58	12	8.01 B
deepsjeng	87	573	87	3	12	15	21	15	19.12	61	38.2 B
imagick	915	18,864	807	40	524	64	113	49	7.01	17	24.1 B
leela	204	1,013	202	15	23	13	17	12	9.34	133	68.7 B
nab	79	730	79	10	25	13	16	13	11.30	18	926 M
xz	149	359	140	3	18	10	18	10	10.17	16	811 M

installed Red Hat Enterprise Linux v7.2, which includes Linux v. 3.10.0-327 as its default kernel.

We compiled and optimized each benchmark using the LLVM [26] compiler toolchain (v. 4.0.1) with -O3. For each experimental run, we execute one (single-threaded) copy of the SPECrate (5xx) version of each benchmark on our otherwise idle machine. Performance results report the mean average execution time of five experimental runs relative to the default running time. To estimate the degree of variability in our results, we also compute 95% confidence intervals for the difference between the means of the default and experimental runs [19], and display these intervals as error bars on the appropriate graphs.

6 EVALUATION

6.1 Length of Calling Context Encoding

We first compare the length of the calling context encodings produced by PCCE, Valence, and HCCE. The "Acyclic Encoding Bits" column of Table 1 shows the number of bits that each approach uses to encode the *acyclic* calling contexts of each benchmark. The "PCCE" column shows the maximum $lg(numCC(n))$ for all n in the original call graph (without back edges). The "Valence" and "HCCE" columns show the maximum number of bits that are necessary to encode the acyclic call graph formed by consolidating each complex SCC into a single node, using either the Valence or PCCE encoding strategy. In other words, the last two columns show the maximum length of the *acycID* for Valence and HCCE, respectively.

Thus, for most benchmarks, each strategy is able to encode the entire acyclic calling context in less than a single word. By comparing the "Valence" and "HCCE" columns, we can see that acyclic PCCE is often more compact than acyclic Valence over the same call graph. However, the difference is only 8 bits, on average, and does not actually require the use of an additional word in most cases. By sacrificing this small amount of compactness, Valence enables scalable encoding and more efficient decoding.

Next, we examine the length of the dynamic calling context encodings generated by each approach. For these experiments, we recorded the length of the calling context encoding (in words) at each function call, and store a count of each length in a histogram that is held in memory throughout the entire run. The line graphs in Figure 4 show the counts for each calling context length for 12 of our 14 benchmarks. We omit *namd* and *lbm*, which do not contain any recursive calls and use only a single word for calling context encoding with all three strategies. Note that the range of the x and y axes are different for each subgraph, and the counts on each y-axis are plotted using a log scale. Additionally, we used these histograms to calculate the average length of the calling context encoding at each function call, and plot these results in Figure 5. For reference, the average and maximum word lengths of the calling context without encoding (i.e., one word for each edge) are also shown under "Runtime Statistics" in Table 1.

The figures show that Valence significantly reduces the length of calling context encodings compared to PCCE, especially for deeper call paths. For example, the *mcf* plot reveals that PCCE requires multiple words for over 80% of its calling contexts (and up to 26 words in the worst case), but Valence uses only a single word to encode the same set of calling contexts. Surprisingly, we found that the worst case encoding length of *gcc* with PCCE is actually longer than the maximum length without encoding. Due to the size of *gcc*'s call graph, PCCE adds four words to the context encoding stack for each recursive call, which results in very lengthy encodings for highly recursive calling contexts.

In most cases, the Valence distribution is more similar to HCCE than PCCE, which suggests that most of the improvement is due to more efficient encoding of recursive calling contexts. However, HCCE does outperform Valence for a few benchmarks, such as *cactuBSSN* and *imagick*, due to its more compact (but less scalable) acyclic encoding. Across all of the benchmarks, the PCCE encoding is about $2.7x$ longer than Valence and more than $3.3x$ longer than HCCE, on average.

6.2 Instrumentation Overhead

Figure 6 shows the execution time of each benchmark with PCCE, Valence, or HCCE instrumentation relative to a default run with no

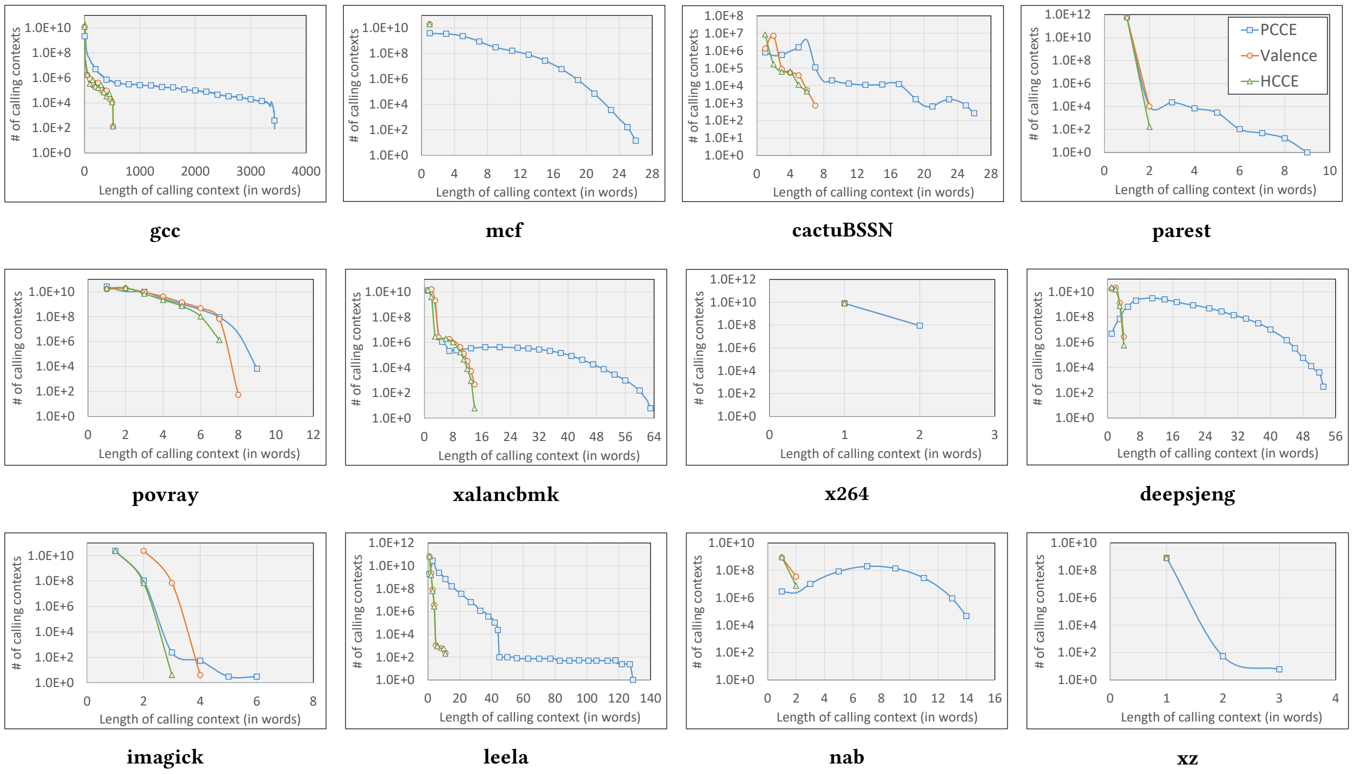


Figure 4: Number of calling context encodings of a particular length (in words) with each encoding strategy. Each plot shows a histogram that counts the lengths of the calling context encodings at each function call.

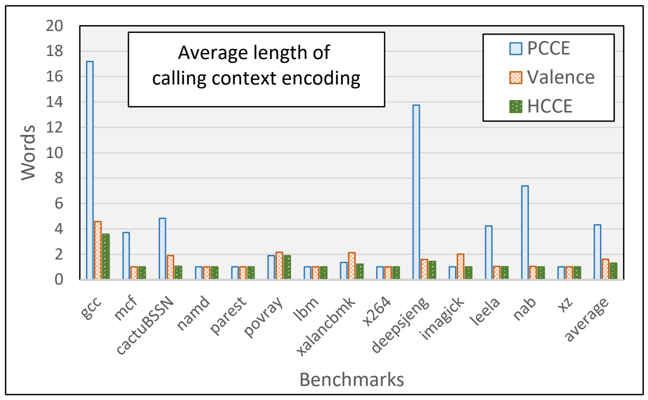


Figure 5: Average length of calling context encoding in words for PCCE, Valence, and HCCE.

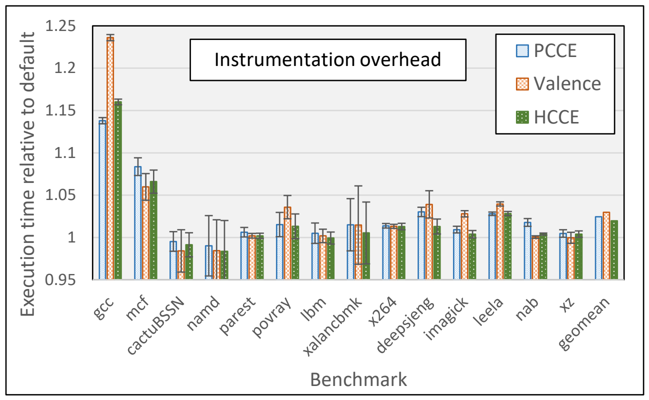
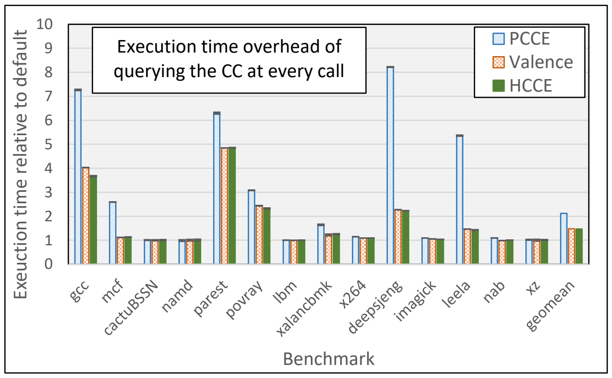


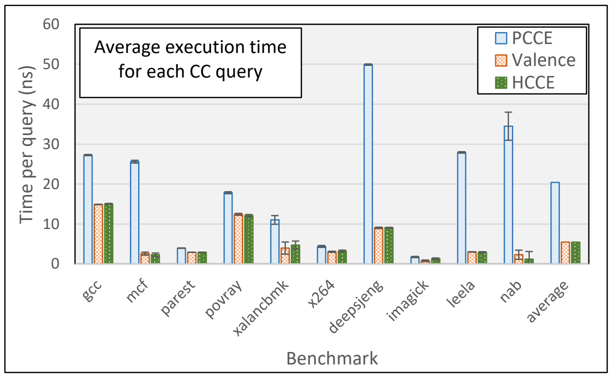
Figure 6: Execution time overhead of call/return instrumentation with PCCE, Valence, and HCCE.

calling context encoding. Overall, the instrumentation overhead is relatively low for all three approaches: less than 10% for all benchmarks except *gcc*, and between 2% to 3%, on average. While the performance variations between the three approaches are typically small, there are design choices that can impact the efficiency of each approach. For instance, Valence and HCCE insert more instrumentation when the call graph has a large number of cyclic edges, while PCCE typically moves more data on and off its context stack.

The clearest discrepancy between the three approaches occurs with *gcc*, which runs about 8% to 10% slower with Valence. However, as noted in Section 5.1, the *gcc* call graph is too large for precise encoding with either PCCE or HCCE, and so, the results shown here underestimate the actual instrumentation costs of these techniques.



a) Total execution time of querying the CC at every function call.



b) Average time (in ns) of each CC query operation.

Figure 7: Execution time overhead of querying the calling context with PCCE, Valence, and HCCE.

6.3 Calling Context Detection Overhead

Lastly, let us consider how a more compact encoding impacts the execution time of applications that read or detect calling context. For this study, we configured our framework to compute the length of the calling context encoding at each function call, similar to the experiments in Section 6.1. However, rather than maintain a histogram, the runtime simply reads and counts each word in the current encoding and then discards it. This setup mimics an *offline* usage scenario, such as a debugging or analysis tool, that reads the calling context frequently, perhaps to enhance understanding, debug, or build a profile of application behavior.

Figure 7(a) displays the execution time of running these experiments with PCCE, Valence, and HCCE, relative to the default execution time for each benchmark. Not surprisingly, querying the context at every function call can incur considerable execution time overheads, especially for benchmarks that make a large number of calls. Some benchmarks, such as *cactuBSSN* and *lbm*, exhibit little or no additional run time overhead due to the relatively low number of function calls they make during the run. For benchmarks with higher overheads, the encoding strategy has a clear and substantial impact on the performance of these experiments. Overall, Valence and HCCE are more efficient than PCCE, and reduce the execution time overhead of these experiments from 2.1*x* to about 1.4*x* of the default running time, on average.

To evaluate the impact of encoding length on execution time more directly, we also used these results to estimate the average cost of a *single* context query. For each benchmark with each encoding strategy, we isolated the total detection time by subtracting the running time with encoding instrumentation alone from the running time when querying the calling context at every call, and then divided the result by the total number of context queries. For this evaluation, we omit four benchmarks (specifically, *cactuBSSN*, *namd*, *lbm*, and *xz*) that make relatively few function calls, and thus, exhibit no significant difference between the running times of the instrumentation-only and context detection experiments.

Figure 7(b) shows the average time cost (in ns) of each calling context query for each benchmark with each encoding strategy. The results indicate a strong correlation between the average length of the context encoding (in Figure 5) and the average run time cost of querying the context. For instance, while Figure 7(a) shows that *nab* runs only about 10% slower with PCCE, the results in Figure 7(b) show that the cost for each context query is several times higher for PCCE than either HCCE or Valence due to a much lengthier encoding. Overall, Valence and HCCE reduce the execution time overhead of querying the calling context by more than 70% compared to PCCE, on average, with these benchmarks.

7 FUTURE WORK

As we take this work forward, our primary goal is to integrate Valence with other compilers, optimizers, debuggers, and analysis tools in order to deliver its benefits to a broad range of real and large scale computing software. In the immediate future, we will apply Valence with FDOs, such as [10] and [32], where the overhead of current context detection techniques can diminish or negate the effectiveness of the optimization. Additionally, while Valence instrumentation costs are relatively low on average, we found that some applications incur slowdowns of more than 20% due to instrumentation alone. Hence, we plan to investigate the use of architectural support for collecting and encoding calling contexts. For example, modern Intel® processors support automated call stack tracking through the use of last branch records (LBRs) [25]. However, these facilities have a limited stack depth (currently, 16 or 32) and require two words for each record on the stack. We plan to examine piecewise use of LBRs to mitigate the depth limitation, and to explore mapping between LBR traces and Valence encoding for efficient instrumentation.

8 CONCLUSION

Many applications generate or rely on calling context information for code profiling, debugging, optimization, and for several other apposite uses of dynamic feedback based adaptation. This work presents Valence, a novel, compiler-based strategy that uses variable length instrumentation to encode program calling contexts efficiently. We have implemented Valence as an LLVM compiler pass, and evaluated it against the state-of-the-art technique for precise calling context encoding (PCCE). Overall, Valence reduces the length of encoding by over 62%, on average, across a standard set of

C/C++ benchmark applications, with negligible impact on instrumentation costs; and therefore, it significantly reduces overheads associated with querying the calling context during execution. Thus, Valence has great potential to improve the efficiency of dynamic optimizations and other applications contingent on calling context feedback.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their thoughtful comments and feedback. This research is supported in part by the National Science Foundation under CCF-1619140 and CCF-1617954 as well as a grant from the Software and Services Group (SSG) at Intel® Corporation.

REFERENCES

[1] 2018. Flang. https://github.com/flang-compiler/flang.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org. *Concurr. Comput. : Pract. Exper.* 22, 6 (April 2010), 685–701. https://doi.org/10.1002/cpe.v22:6

[3] Glenn Ammons, Thomas Ball, and James R. Larus. 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)*. ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/258915.258924

[4] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz. 2007. Stack Trace Analysis for Large Scale Debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–10. https://doi.org/10.1109/IPDPS.2007.370254

[5] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 46–57. http://dl.acm.org/citation.cfm?id=243846.243857

[6] David Binkley. 1997. Semantics Guided Regression Test Cost Reduction. *IEEE Trans. Softw. Eng.* 23, 8 (Aug. 1997), 498–516. https://doi.org/10.1109/32.624306

[7] Michael D. Bond, Graham Z. Baker, and Samuel Z. Guyer. 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)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1806596.1806599

[8] Michael D. Bond and Kathryn S. McKinley. 2007. Probabilistic Calling Context. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 97–112. https://doi.org/10.1145/1297027.1297035

[9] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. 2007. Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 405–422. https://doi.org/10.1145/1297027.1297057

[10] Milind Chabbi, Wim Lavrijsen, Wibe de Jong, Koushik Sen, John Mellor-Crummey, and Costin Iancu. 2015. Barrier Elision for Production Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 109–119. https://doi.org/10.1145/2688500.2688502

[11] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False-sharing Detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 152–167. https://doi.org/10.1145/3178487.3178499

[12] Trishul M. Chilimbi and Vinod Ganapathy. 2006. HeapMD: Identifying Heap-based Bugs Using Anomaly Detection. *SIGPLAN Not.* 41, 11 (Oct. 2006), 219–228. https://doi.org/10.1145/1168918.1168885

[13] James Clause and Alessandro Orso. 2010. LEAKPOINT: Pinpointing the Causes of Memory Leaks. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 515–524. https://doi.org/10.1145/1806799.1806874

[14] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2016. Mining Hot Calling Contexts in Small Space. *Softw. Pract. Exper.* 46, 8 (Aug. 2016), 1131–1152. https://doi.org/10.1002/spe.2348

[15] T. Chad Effler, Adam P. Howard, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, and Prasad A. Kulkarni. 2018. On Automated Feedback-Driven Data Placement in Hybrid Memories.. In *LNCS International Conference on Architecture of Computing Systems (ARCS '18)*.

[16] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (SP '03)*. IEEE Computer Society, Washington, DC, USA, 62–. http://dl.acm.org/citation.cfm?id=829515.830554

[17] Eclipse Foundation. 2018. "Eclipse IDE for Java Developers". https://www.eclipse.org/downloads/packages/release/photon/r/eclipse-ide-java-developers

[18] Free Software Foundation. 2018. "GDB: The GNU Project Debugger". https://www.gnu.org/software/gdb/

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Object-oriented programming systems, languages, and applications (OOPSLA '07)*. 57–76.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[21] Rentong Guo, Xiaofei Liao, Hai Jin, Jianhui Yue, and Guang Tan. 2015. Night-Watch: integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 307–318.

[22] Qualcomm Innovation Center Ivan Baev. 2015. Profile-Based Indirect Call Promotion. https://llvm.org/devmtg/2015-10/slides/Baev-IndirectCallPromotion.pdf.

[23] Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer Memory Management for Managed Language Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 488–504. https://doi.org/10.1145/2814270.2814322

[24] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. 2014. Automated Memory Leak Detection for Production Use. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 825–836. https://doi.org/10.1145/2568225.2568311

[25] Andi Kleen. 2016. Advanced usage of last branch records. https://lwn.net/Articles/680996/

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[27] Jianjun Li, Zhenjiang Wang, Chenggang Wu, Wei-Chung Hsu, and Di Xu. 2014. Dynamic and Adaptive Calling Context Encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 120, 12 pages. https://doi.org/10.1145/2544137.2544167

[28] Xu Liu and John Mellor-Crummey. 2011. Pinpointing Data Locality Problems Using Data-centric Analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 171–180. http://dl.acm.org/citation.cfm?id=2190025.2190064

[29] Xu Liu and Bo Wu. 2015. ScaAnalyzer: A Tool to Identify Memory Scalability Bottlenecks in Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 47, 12 pages. https://doi.org/10.1145/2807591.2807648

[30] Linux Programmer's Manual. 2017. backtrace, backtrace_symbols, backtrace_symbols_fd - support for application self-debugging. http://man7.org/linux/man-pages/man3/backtrace.3.html

[31] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[32] Matthew Benjamin Olson, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, M. Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated Application Guidance for Hybrid Memory Systems.. In *IEEE International Conference on Networking, Architecture, and Storage (NAS '18)*.

[33] Atanas Rountev, Scott Kagan, and Jason Sawin. 2005. Coverage criteria for testing of object interactions in sequence diagrams. In *In Fundamental Approaches to Software Engineering, LNCS 3442*. 282–297.

[34] H. Servat, A. J. Peña, G. Llort, E. Mercadal, H. Hoppe, and J. Labarta. 2017. Automating the Application Data Placement in Hybrid Memory Systems. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*.

[35] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. 2015. Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 401–413. https://doi.org/10.1145/2810103.2813654

[36] SPEC. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[37] William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2010. Precise Calling Context Encoding. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 525–534. https://doi.org/10.1145/1806799.1806875

[38] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/SC.2010.47

[39] R. Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 114–121. https://doi.org/10.1109/SWAT.1971.10

[40] Dave Watson, Arun Sharma, and David Mosberger-Tang. 2017. The libunwind project. https://www.nongnu.org/libunwind/index.html

[41] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. REDSPY: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 47–61. https://doi.org/10.1145/3037697.3037729

[42] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 332–347. https://doi.org/10.1145/3173162.3177159

[43] Qiang Zeng, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Peng Liu. 2014. DeltaPath: Precise and Scalable Calling Context Encoding. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 109, 11 pages. https://doi.org/10.1145/2544137.2544150

[44] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. 2006. Accurate, Efficient, and Adaptive Calling Context Profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 263–271. https://doi.org/10.1145/1133981.1134012