Lucce: Lazy and Unintrusive Calling Context Encoding for Java

Nikita Salnikov-Tarnovski and Vesal Vojdani

Department of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia
nikem@plumbr.eu,vesal@ut.ee

Abstract. A stack trace provides detailed information about an event's location during the execution of a program. Many activities in software engineering, such as debugging, troubleshooting, and performance optimization, benefit from accurate location information, but keeping track of the entire call stack can be too expensive. There has been much progress recently on efficient calling context encoding schemes that work well despite the dynamic nature of the call graph of JVM-based applications. However, existing tools for collecting such information either do not guarantee accurate reconstruction of the entire stack trace or are hard to integrate into unintrusive monitoring tools. Our proposed encoding scheme, Lucce, computes accurate calling context information and can reconstruct the entire trace from essentially a single integer per event, even in the presence of dynamic class loading and recursion. As we do not rely on static analysis or any advance processing of the application code, Lucce can be be easily integrated into any monitoring tool and used by application developers without much hassle.

1 Introduction

When an application fails on the JVM, the run-time system produces a stack trace with the entire sequence of calls leading up to the offending program location. Knowing the full calling context is critical when debugging object-oriented applications due to their high level of indirection and reuse. Producing a single stack trace upon program failure is easy enough, but many monitoring tools run alongside an application within the same JVM and continuously watch for particular events of interest in the host program. Whenever such events occur, the tool records or processes those events. Eventually, it may have identified that an event was, e.g., the root cause of a memory leak and wishes to report this to the user.

In this setting, generating the whole stack trace and storing it with every event is no longer a realistic option. Unwinding the stack is quite an expensive operation on the JVM, as it linearly depends on the depth of the current execution stack. When events of interest are very frequent and/or short-lived, such as new object creation or lock contention events, this additional work can lead to about a tenfold overhead during program execution. Instead, we would like the

location-specific information to be readily available for monitoring tools to use without requiring extra work during event recording. As many tools separate the process of recording events from their analysis and eventual presentation to end users, we would like to store as little location-specific information as required to faithfully restore the full stack trace leading to any particular recorded event.

In this paper, we present Lucce (rhymes with "ducky"), a calling context encoding scheme that represents every call stack encountered during program execution as a single number, which can later be fully decoded to the original. In our implementation, this pre-computed number is available as the value of a local variable for any method in the running program, making it convenient for clients of our framework to grab and record along with their own event-related information. Decoding it to the actual stack trace may happen later, either off-site, after the program has already terminated, or in parallel with the program's execution, in a separate thread outside critical execution paths.

Our solution, Lucce, extends the *precise calling context encoding* (PCCE) of Sumner et al. [15] to deal with the dynamic nature of JVM call graphs. We lazily build the annotated call graph required by PCCE as we discover new edges during program execution. Thus, whenever the call graph changes, rather than recomputing annotations for the entire graph, we only recompute values for nodes that are subsequently visited. We do not necessarily have a correctly annotated call graph in memory at all times; nevertheless, we can guarantee that any call context visited during program execution and encoded by Lucce can be correctly decoded to recover the actual stack trace. This decoding is possible because we log graph changing events. In summary, this paper makes the following contributions:

- We provide an extension to the PCCE scheme [15] that can correctly and uniformly deal with the dynamic class loading, virtual method dispatch, and callbacks from native code.
- We show that this modification leads to a natural treatment of recursive calls without additional run-time storage requirements, as the burden is offloaded to a write-only event log. We prove that all generated encodings can be decoded back to the original call context.
- We have implemented this approach in a proof-of-concept software package and evaluated its runtime characteristics using the Dacapo benchmarking suite.

There are naturally other approaches to call context encoding. DeltaPath [19], DACCE [13], and Breadcrumbs [9] also address the issue of changing call graphs. We try to do them justice in Section 7, but as far as we can tell, parallel construction of *calling context trees* [3, 6] is the only alternative that produces an accurate stack trace in the presence of dynamic class loading and callbacks from native code without requiring modifications to the JVM itself. Compared to Lucce, this method maintains the CCT in memory, which can grow much larger than than the call graph maintained by our algorithm. The more recent work on calling context uptrees [12], though, requires extensive modifications to the runtime environment.

The target application for our method is within deployable monitoring tools, i.e., tools that track an enterprise Java application in a production setting for a significant amount of time. The application we have primarily in mind is memory leak detection based on statistical analysis [11, 18]; however, the general characteristics for which our approach is suitable are as follows:

1. Monitoring is transparent to the user: our instrumentation should not significantly influence the behaviour of the application under scrutiny. It should not incur high performance overhead, and when looking for memory leaks, we cannot let the context encoding itself consume unbounded amounts of runtime memory.
2. Tracked events are numerous: the client analysis records many context-sensitive events online, such as object allocation, but only a subsequent analysis of the logs may reveal which events are relevant. This is the main motivation for wanting as compact an encoding as possible.
3. Reconstruction of calling context information can occur offline and off-site: having identified the few relevant events, it is acceptable if reconstruction of stack traces is more time consuming. It is far more important that logs are not bloated since we may need to store and transfer them.
4. Imprecision is *not* acceptable: we would be sorely disappointed if after a lengthy monitoring process and statistical analysis, we cannot reconstruct the stack trace for the particular events that are deemed relevant. We need a valid stack trace, as good a trace as produced by the JVM itself, also in the presence of dynamically loaded classes or callbacks from native code.
5. Call contexts can be added unintrusively: developers of profiling tools can add support for call contexts without too much effort and, more importantly, without impacting the usability and portability of their monitoring tool. Thus, we must be able to run the tool on commercial JVMs and require as few changes to the client application as possible.

The first three are general properties of deployable bug detectors, as described by Bond et al. [9], but when considering all requirements together, we see our proposal as the best fit.

The rest of the paper is organized as follows: In Section 2, we briefly describe the standard PCCE encoding algorithm and introduce the necessary notation. In Section 3, we consider an example of a dynamic call graph and give an intuitive description of our solution. In Section 4, we describe how programs are instrumented in order to track method invocations, and in Section 5, we present our encoding algorithm and prove its correctness. We present the experimental results in Section 6 and related work in Section 7.

2 Preliminaries

The ultimate goal of a call context encoding algorithm is to provide monitoring tools with the encoding of any calling context active during the execution of a program as a single number, called henceforth the *calling context identifier*

or callId, which can be used to uniquely identify that call context. As we are interested in displaying the stack trace to the user, it should be possible to decode any given callId and regain the stack trace that generated it.

A *call graph* (CG) is a pair $\langle N, E \rangle$ where N is a set of nodes with each node representing a method and E is a set of directed edges. Each edge $e \in E$ is a triple $\langle p, n, l \rangle$, in which $p, n \in N$, represent a caller and callee, respectively, and l represents the call site where p calls n. Here, call edges are modeled as a triple instead of a caller-callee pair because we want to model cases in which a caller may have multiple invocations of the same callee. We introduce the following operators to move around in the call graph:

$$\begin{aligned}
\langle p, n, l \rangle^{\rightarrow} &= p && \text{(source of an edge)} \\
\langle p, n, l \rangle^{\leftarrow} &= n && \text{(target of an edge)} \\
n^{-} &= \{e \in E \mid e^{\leftarrow} = n\} && \text{(incoming edges of a node)} \\
p^{+} &= \{e \in E \mid e^{\rightarrow} = p\} && \text{(outgoing edges of a node)}
\end{aligned}$$

For a given invocation of a method n, the *calling context* is a path, i.e., a sequence of edges in the CG, leading from the root node *root* to the node representing n. We use the notation $\pi \colon p \to q$ to denote a path from p to q. Let CC be the set of all possible calling contexts, and let CC_n denote those contexts ending with invocations of n. A valid calling context encoding scheme is a one-to-one function $En \colon CC \to \mathbb{N}$ that maps given path in the call graph to a single natural number.

To arrive at the PCCE algorithm [15], upon which Lucce is built, we first define the function $numCC : N \to \mathbb{N}$, such that $numCC(n) = |CC_n|$; i.e., we count the total number of contexts for a fixed destination node n. This can easily be computed by traversing the static call graph:

$$numCC(n) = \begin{cases} 1 & \text{if } n \text{ is a root node of CG} \\ \displaystyle\sum_{e \in n^{-}} numCC(e^{\rightarrow}) & \text{otherwise} \end{cases}$$

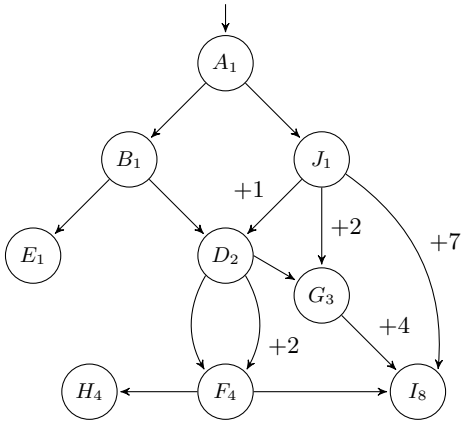
We impose a fixed ordering on the elements of $n^{-} = \{e_1, e_2, \ldots, e_{|n^{-}|}\}$, so each edge e_i can be annotated with an *addition value*:

$$av(e_i) = \sum_{j<i} numCC(e_j{}^{\rightarrow}).$$

These are used to effectively partition CC_n based on the incoming edges. While making calls during program execution, the value of the current callId is increased by $av(e)$ when traversing an edge e from caller to callee. On returning from the callee back to the caller, the value of the current callId is decreased by $av(e)$. For any active call context encountered during execution, we have computed the following encoding:

$$\begin{aligned}
En([\,]) &= 0 \\
En(e : \pi) &= av(e) + En(\pi)
\end{aligned}$$

where [ ] denotes the empty path, and $e : \pi$ describes a path starting with e followed by the path π.



Context	callId
AB	0
AJ	0
ABD	0
AJD	1
ABE	0
ABD^1F	0
AJD^1F	1
ABD^2F	2
AJD^2F	3
...	

Fig. 1: Example of graph encoding [15].

Example 1. Figure 1 demonstrates the encoding process. Each node is subscripted with its *numCC* value, while edges are annotated with addition values where they differ from 0. The table provides a few examples of encodings for various call contexts in the call graph. In this example, D calls F at two different call sites; these a distinguished by their superscripts, D^1 and D^2.

Given an encoded callId c leading to a method n, we can reproduce the stack trace one step at a time by considering the addition values of the method's incoming edges n^-. The previous edge in the stack trace is the incoming edge with the greatest addition value not exceeding c:

$$prev(n, c) = e \iff av(e) = \max\{av(e) \mid e \in n^-, av(e) \leq c\}$$

Thus, we define the decoding function:

$$De(n, c) = \begin{cases} [\,] & \text{if } n = root \\ De(e^{\rightarrow}, c - av(e)) : e & \text{otherwise; where } e = prev(n, c) \end{cases}$$

We have overloaded the list operator and write $\pi : e$ to place the edge at the end of the list. The path is decoded backwards, retracing the steps of the execution.

This is a valid encoding scheme and the decoding is precise; that is, for any path π in the call graph leading to a method n, we can successfully decode its encoding:

$$De(n, En(\pi)) = \pi$$

3 Dealing with dynamic call graphs

The PCCE approach, which assumes that the whole call graph is known in advance, cannot be transfered verbatim to the situation of a typical Java application. The JVM uses a dynamic class loading mechanism to load parts of the application into memory only when they are executed for the first time. As a result, the call graph of an application can change at any moment of the program's execution.

Example 2 (The Problem). Let us consider the example in Figure 2. Given this call graph, the execution path $A \to B \to D \to E$ will be encoded as the integer 1. At some point later in the program's execution, a new edge appears, $X \to B$. If we were to decode the calling context value, 1, starting from method E, we would get the erroneous trace $X \to B \to C \to E$.

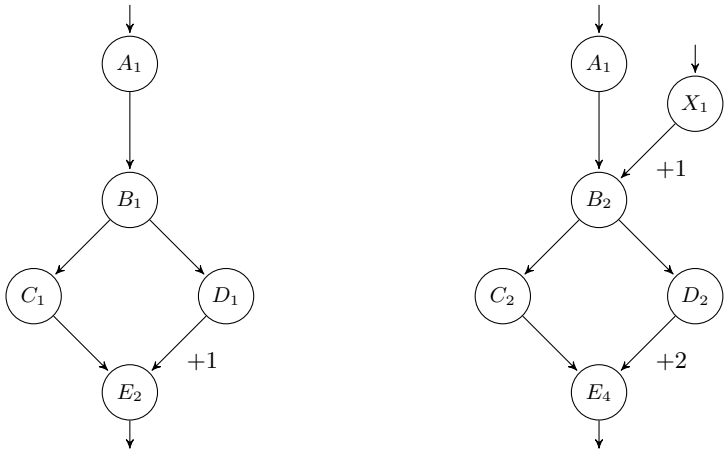


Fig. 2: Call graph before and after discovering a new edge.

Our decoding can go wrong because for any given node $n \in N$, the value of $numCC(n)$ may change when new edges emerge in the graph. This in turn leads to changes in the addition values and thus to different results for the encoding and decoding algorithms. We will describe our solution to the general problem in Section 5; here, we aim to illustrate how our encoding algorithm deals with Example 2.

As our algorithm is dynamic, we must consider a slightly longer execution to build an equivalent call graph. The three intermediate states worth considering are depicted in Figure 3. We first assume the program calls the following sequence of methods: $A \to B \to C \to E$. Our algorithm updates the $numCC$ counters and the addition values on the fly, and for this first singleton path, the encoding

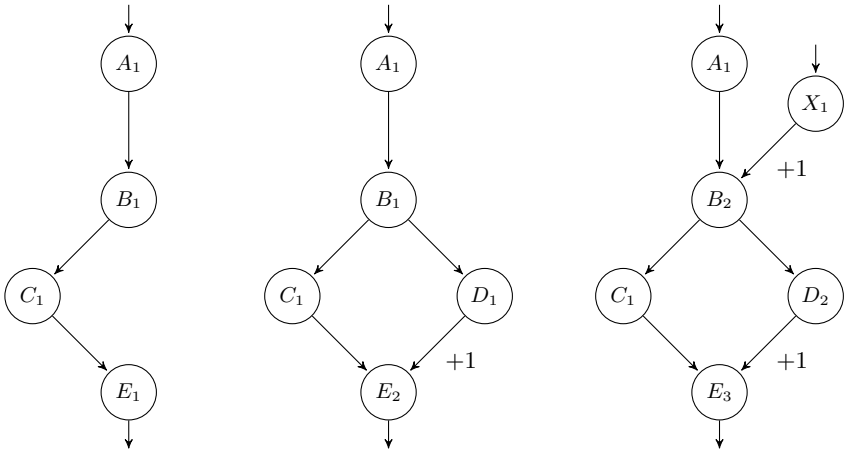


Fig. 3: Lucce's dynamically generated call graphs.

is obviously zero. Whenever an edge is added to our graph, we log this as an edge adding event and increment our version counter. The encoding for this particular path, then, is not just zero, but the pair $\langle 0, v_1 \rangle$.

Then, with E and D returning and being popped off the call stack, execution resumes at B which immediately calls D. When D in turn calls E, we have the path $A \to B \to D \to E$. As our algorithm now enters E again and computes the addition value for the new edge, we obtain the second graph in Figure 3 where this context is encoded as $\langle 1, v_2 \rangle$.

Finally, assume all methods return all the way to the main method, or alternatively, assume another thread takes the path $X \to B \to D \to E$. As we now enter each node, the values are updated, and we end up encoding this path as $\langle 2, v_3 \rangle$. The instrumentations for the final graph are not identical to what we had in Figure 2; in particular, the new graph does not count the path $X \to B \to C \to E$. However, the paths that we actually did visit can be decoded in the recorded versions of the call graph.

4 Instrumentation of bytecode

Before looking at the encoding algorithm, it helps to see the context in which the encoding takes place. We have implemented the algorithm as a JVM Tool Interface agent [14] that instruments the bytecode of an arbitrary Java application using the ASM bytecode manipulation library [10]. The following modifications are made to the bytecode of all loaded classes:

– Two new local variables are added to every method of every class. One of these variables is used to store the current value of the callId during a method's execution, while the other stores the current version of the call

graph. As we saw in the previous section, this pair together constitutes our calling context encoding.
- As the first instruction of every method, the call to the algorithm's *Entered* function is inserted. This call passes the unique identifier of the entered method. The method will return the new value of callId and the graph version, which will be stored in the local variables mentioned above.
- Before every method call bytecode instruction, a call to the algorithm's *Calling* function is inserted. This call passes the unique identifier of the call site, where this method call originates, and the current value of the callId.

Example 3. For illustration, we show a source-to-source transformation that achieves the intended effect of our instrumentation. Consider the following recursive method: **int** $f()\{\dots f^l()\dots\}$, where we have annotated the call site with its location l. The method is transformed into the following pseudo-code:

$$\textbf{int } f()\{\text{callId } \langle c, v\rangle = \mathit{Entered}(f); \dots \mathit{Calling}(l, \langle c, v\rangle); f^l(); \dots\}$$

For this example, our abstract callId type includes the version as well.

We update the callId only from within the called method, not before an actual invocation, because at the time of method invocation, it is generally impossible to determine which method will be called. This information is available only after JVM has performed method resolution. Trying to hook into this process is error prone, and with the introduction of MethodHandlers in JDK7, the JVM now has a whole new way to perform method resolution. It would require much extra work to adapt to this new mechanism. Even worse, when native code call Java methods through the Java native interface, we cannot instrument the call site at all. Therefore, it is much more convenient to notify the algorithm about a new method when it has already appeared on the call stack.

However, this necessitates passing information about the caller into called method. This is exactly the job of the *Calling* method mentioned above. It must precede each method invocation in order to store the caller information to be consumed when we enter the new method. For native code, we associate the location with the invocation of the native method, producing stack traces such as: `main`(`File.java : 13`); `f`(`native method`); `h`(`File.java : 42`), which is the best we can do for native code calling Java methods.

5 Dynamic calling context encoding

Having instrumented the program, the encoding algorithm receives sequences of calls to the internal functions signaling method entry and exit; meanwhile, critical information is stored in the local variables of instrumented methods. In this way, we do not actually subtract the addition value when returning from a method call because execution resumes with the previous values of the locals. Further, as we will prove in this section, despite the seeming complexity of the run-time behavior, our encoding is directly invertible due to the logging of graph

versions. We can therefore model the execution of an instrumented program as again generating a sequence of edges.

Whenever we detect that calling graph has changed, we record the corresponding event (of adding or changing a node or an edge in the graph). The encoding part of the algorithm then can always operate on the currently actual graph as if no changes have occurred. With every recorded value of callId, we store the version of the graph that was valid at the time of this encoding. Then, when we will be decoding this value, we will be able to "replay" all prior graph changing events. As we decode, we further ensure that each node is rewound to the graph version in which its addition values were computed.

For the presentation, we model our even history by assuming that we have persistent versions of the dynamic call graph for all versions at hand. The changing data structure is then a triple $G = \langle E, numCC, lv \rangle$, where E is the set of edges between methods, as before. Although the set of nodes N can also change dynamically as new classes are loaded at runtime, we may simplify the model. In reality, we assign methods unique identifiers, and since this mapping is global and valid for all graph versions, we can for clarity of presentation, assume we have a fixed set of methods. The $numCC$ mapping serves the same role as in the standard PCCE encoding, while lv is a mapping from nodes to graph versions. It maps a node p to the graph that was valid when we last called a method from p. This mapping is useful for optimization because we can cache the computation of addition values and avoid recomputing if we can ensure that the graph has not changed since last visiting a node. Here, we are concerned about correctness, and this mapping plays a critical role in ensuring correct decoding.

Given a call path π, we compute the encoding edge by edge starting from the encoding pair $\langle 0, 0 \rangle$:

$$
\begin{aligned}
&En(\pi) &&= En'(\pi,\ \langle 0, 0 \rangle) \text{ where} \\
&En'([\,], &&cv) = cv \\
&En'(e : \pi, &&cv) = En'(\pi, Update(e, cv))
\end{aligned}
$$

Upon traversing an edge e we update the call graph, recording graph changing event if necessary, and we encode the current call path by adding the addition value of e to the current value c. Algorithm 1 returns the new callId while updating the call graph.

In this high-level presentation of the algorithm, we have abstracted the details of graph versioning under the carpet. The notation $x \leftarrow y$ should therefore be interpreted as doing all the important work for us; i.e., it evaluates y in the current version of the shared structure, checks if x changes, and if it does, increments the graph version and logs the update. Note that the current version of the graph when entering this method is *not* necessarily v. This is the graph version that was valid when entering the parent method p, but that method may have called many other methods and only then proceeded to take the call edge we are currently encoding. There is no simple relationship between v and v', which is why we store v in $lv(p)$.

Algorithm 1 Updating the call graph

Input: call edge e, current callId c, and graph version v. The algorithm accesses the graph G as a global data structure.

Output: The new callId and graph version for the callee. Additionally, G is potentially updated and its version number incremented.

function UPDATE(e, cv)
- let $\langle p, n, l\rangle = e$ and $\langle c, v\rangle = cv$ ▷ Decompose the arguments.
- $lv(p) \leftarrow v$ ▷ Backup the graph version of the caller.
- $E \leftarrow E \cup \{e\}$ ▷ Add the new edge if needed.
- $numCC(n) \leftarrow \sum_{e \in n^-} numCC(e^{\rightarrow})$ ▷ Recompute the callee's *numCC*.
- let $v' \leftarrow$ current version of G ▷ Get version from the global structure.
- **return** $\langle c + av(e), v'\rangle$ ▷ $av(e)$ is computed in the updated graph.

end function

This takes us to one further implementation detail we must point out: it is critical for correctness in a multi-threaded setting that the updates and the query to obtain the graph version all happen within the same atomic block; we need $lv(p)$ and the computed addition value to have the same values in the graph v' as they had during this update to the graph. As we now turn to the decoding, we see that this relationship is effectively the correctness invariant of the algorithm, so we cannot allow the graph to be changed in-between.

In order to obtain the whole chain of call sites that have led to an event of interest, we have to decode the recorded value of callId. For this, we need to compute addition values in a given version of the call graph $G_v = \langle E_v, numCC_v, lv_v\rangle$. The definitions of $av(e, v)$ and $prev(e, c, v)$ are essentially the same as in Section 2, but take an extra parameter referring to the version of the graph in which they are evaluated. With this in mind, we can define the decoding:

$$De(n, \langle c, v\rangle) = \begin{cases} [\,] & \text{if } n = root \\ De(e^{\rightarrow}, \langle c - av(e, v),\ lv_v(e^{\rightarrow})\rangle) : e & \text{otherwise} \end{cases}$$
$$\text{where } e = prev(n, c, v)$$

As we step back from n, we hopefully obtain the previous callId by $c - av(e, v)$ and from $lv_v(e^{\rightarrow})$ we should have the version of the call graph that was valid when the caller was entered. We now prove that this in fact is the inverse of the encoding step.

Theorem 1. *If execution reaches a method n with the call stack π: root $\rightarrow n$, the call identifier produced by our algorithm can be decoded to recover π, i.e., $De(n, En(\pi)) = \pi$.*

Proof. We prove this by showing that decodability is a preserved invariant during each step of the encoding process. Initially, we have an empty call stack starting and ending at the *root* node, which is trivial to decode: $De(root, En([\,])) = [\,]$.

To prove that decodability is preserved, we decompose $\pi = \pi' : e$, and assume that for π': *root* $\rightarrow p$, we generated the encoding $\langle c, v\rangle$ which correctly decodes

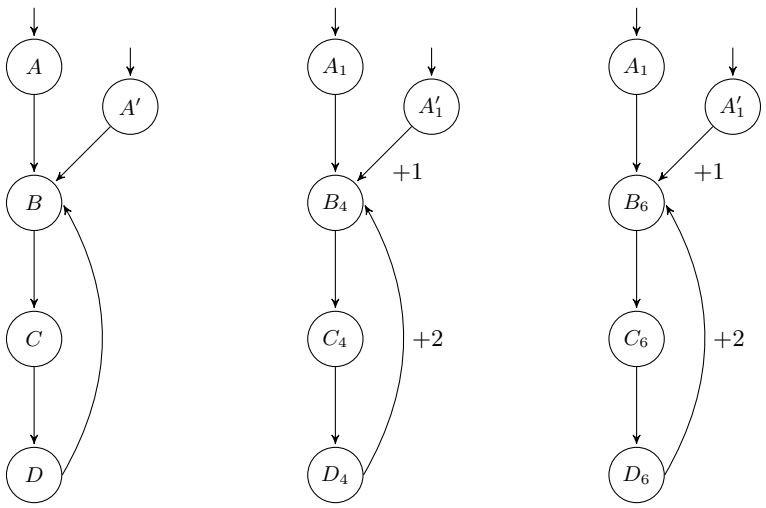


Fig. 4: Treatment of recursion

back to π'. As we then traverse an edge e, we update the graph and produce the encoding $\langle c + av(e, v'), v' \rangle$. This too will be correctly decoded because the decoding algorithm computes $prev(n, c, v')$ to obtain e and subtracts $av(e, v')$ to recover c. However, this callId was valid in the graph v, so our decoding algorithm rewinds the graph to $lv_{v'}(e^{\rightarrow})$.

Thus, for any two consecutive stack frames, say method p entered with $\langle c, v \rangle$ and method n entered with $\langle c', v' \rangle$, our encoding algorithm must preserve $v = lv_{v'}(n)$. This is first enforced by the function *Update* upon entering n, and since v and v' are persistent graph versions, this relationship is invariant. □

This proves that everything is in order, and recursion should simply work; still, it may be interesting to see what the algorithm actually does when there are recursive calls in the program. Previous work on extending PCCE retained the original approach of limiting the basic algorithm to acyclic call graphs only. For call graphs with cycles, which corresponds to the recursive calls in the original application, the recursive call is pushed onto a local stack, resulting in a stack of encodings of acyclic subpaths. In contrast, we happily apply the original idea to cyclic subgraphs as well.

Example 4 (Recursion). Consider the leftmost call graph in Figure 4. Here, $D \rightarrow B$ is a recursive back edge. Now, if we try to encode calling context $A' \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C$, disregarding the fact that $D \rightarrow B$ is recursive, we obtain the addition values that are annotated in the second graph of the figure. When we reached B for the second time, we simply applied the above definitions, and computed $av(DB) = numCC(A) + numCC(A') = 2$. Thus we return callId $= 3$ for calling context $A' \rightarrow B \rightarrow C \rightarrow D \rightarrow B \rightarrow C$ and this can safely be decoded back into the correct calling context.

If the recursion continues, we too continue to follow the Algorithm 1, and whenever the value of $numCC$ for some caller has changed, we update the value of $numCC$ for the callee as well. And we continue doing so as we follow the call sequence. This results in the $numCC$ values of nodes B, C, D changing with each level of recursion, which unfortunately leads to many "nodeUpdated" logging events, but we have learned to live with this trade-off.

6 Experimental results

We have evaluated our implementation on the DaCapo benchmark suite [7] consists of a set of open source real world applications with non-trivial workload, object allocation rates, and memory consumption. For analysis of the current paper's algorithm DaCapo version 9.12, released in 2009, was used. In our test runs we had to exclude two benchmarks, *tradebeans* and *tradesoap*, as they often failed with runtime errors. We have run all benchmark using default settings and default load size. The overhead factor reported is obtained as follows:

- Run benchmark 31 times in row, using "-n 31" command line option for DaCapo
- Discard first run, as its results always deviates from others by large margin, due to initial JVM optimizations cost and startup overhead
- Collect resulting 30 reported running time and calculate average running time.

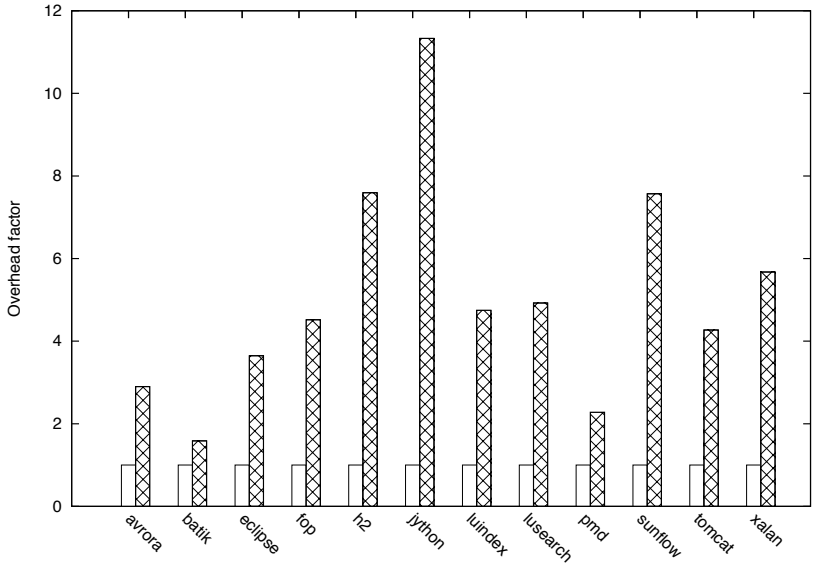


Fig. 5: Experimental evaluation of overhead.

Each benchmark was run first without our agent attached and then together with it. The average running time of the latter was divided by the former and the resulting ratio was reported in Figure 5. The first bar is simply the reference run — its value is always one; the second bar shows the overhead factor of computing the callId. The overall overhead is high, much higher than we would like. However, it is not clear if we could be much faster while still being able to produce stack traces in all cases.

7 Related work

The original idea of encoding a path as a single number based on the different ways to arrive at a given location was introduced for the intra-procedural case by Ball and Larus [5]. This was not directly transferable to the inter-procedural case. The necessary modifications were made by Sumner et al. [15]. The key observation of the authors was that in order to provide context sensitivity, there is no need to have an unique context identifier for every calling context in the application. As we are interested in the calling context for some event, which has happened in some point in the application, it will be sufficient if "all unique paths leading from the root to a specific node have unique encodings, because we only need to distinguish the different contexts with respect to that node".

However, this method analyze the source code of the whole program in advance to construct the complete call graph that enables the encoding of inter-procedural paths. The performance characteristics of this approach is very impressive: an average overhead of about $2-4\%$ is reported, with worst case overhead reaching 10%. As this approach does not use any runtime data structure, memory overhead, although not reported by the authors, can be assumed to be non-existing. Unfortunately, these numbers cannot be compared directly with the ones reported in this article as benchmark applications differ significantly. The major drawback of their approach is the inability to efficiently cope with virtual functions and dynamic class loading, which are very common characteristics of Java applications.

DeltaPath, presented in [19], tried to address these issues. In the case of dynamic class loading, the authors detect in the runtime that execution has arrived to some node from an unexpected (not seen during static program analysis) caller. Then current call context value is then pushed onto the stack together with a flag marking the specific reason the value was pushed (dynamic class loading in this case), and encoding resumes with the value 0. This allows the construction of a valid partial trace, leaving out the dynamically loaded portion.

The implementation of DeltaPath consists of static bytecode instrumentation, which has to happen before profiling session, and dynamic java agent for monitoring and instrumenting dynamically loaded classes using the information obtained by static part. Implementation is compatible with stock JVM compatible with Java 5 or later. The extensive usage of stack data structures for holding intermediate calling context values can make reporting the current call context too expensive, as on each profiling event recording we have to pass the whole

stack as part of the event metadata. The average reported depth of those stacks is between 1 and 4.4, with maximum value of 26. So on each profiling event recording we have to copy up to tens of values. For 11 out of 15 benchmarks in SpecJVM suite, the authors report average slowdown of 6.5%. But due to a huge impact in other benchmarks, the overall average slowdown is reported to be 32%. There is no memory overhead reported.

One of the drawbacks of DeltaPath is the necessity to analyze and instrument the source code of the application before it runs, thus incurring extra burden on DeltaPath users. A more pressing concern is that the traces for dynamically loaded code, although correct for the previously seen portions of code, nevertheless do not contain calls to newly loaded methods. This issue is addressed by DACCE (Dynamic and Adaptive Calling Context Encoding) [13]. DACCE dynamically constructs actual call graph of the program execution during run time, by adding each node and edge when they are first encountered. Edges are not encoded nor instrumented at once. Instead, periodically, when some inner thresholds are reached, DACCE stops the execution of the whole application, analyses the entire graph known to this moment, and encodes all known edges based on the current graph. During this encoding process some optimizations are applied as well; e.g., encoding the hottest edges with zero, thus effectively removing any instrumentation along this edge. All calling context ids collected before the re-encoding process either should be re-encoded and updated or augmented with versions of the graph, when they were encoded.

As call graph is re-encoded on each graph update there are some edges in the call graph may not yet be instrumented. DACCE handles them in the similar way as DeltaPath by pushing current encoding values to a stack. From the point of view of the user of the calling context information, DACCE is identical to the DeltaPath. The authors measured the runtime overhead of their implementation using SPEC CPU2006 benchmark [1]. The reported average runtime overhead is about 2% with maximum being below 8%. There is no memory overhead reported.

Another approach to encoding calling contexts is the probabilistic calling contexts (PCC) due to Bond and McKinley [8]. The idea is to compute an evenly distributed random value to provide a probabilistically unique identifier for each calling context. In the original article, the PCC value is an accumulated hash value of the entire stack trace that the analyzer can use to distinguish between different calling contexts. The main drawback of this approach is the lack of decoding capabilities. PCC does provide a way to distinguish between two calling contexts and to identify an already known calling context, but it does not allow us to reconstruct the exact calling context.

This ability was provided by Bond et al. [9] as the Breadcrumbs tool. They use PCCs to record calling context and then store additional information that allows them to decode a PCC back to a sequence of function calls. In order to enable the The additional information facilitates an iterative backward search among all possible candidate call sites. Breadcrumbs is implemented on the Jikes RVM [2] by instrumenting every call site of the application with PCC calcula-

tion and, possibly, with recording all PCC values generated at this callsite. In addition to that, partial call graph is constructed by augmenting the just-in-time compiler. Decoding is performed offline, after the application runtime, during a separate reporting phase. The authors use the Dacapo benchmark suite [7] for evaluation of the runtime overhead. For a configuration that only computes PCC values, without collecting any additional information, an overhead of around 5% is reported. Recording every generated PCC value on every call site, which is required for precise PCC decoding, results in the overhead raising to 100% on average, ranging from around 20% up to more than 160% (the authors do not provide exact numbers) depending on the test application.

A completely different approach for providing the calling context information is to construct a *Calling Context Tree* (or CCT for short). This data structure was first presented by Ammons et al. [3] as an efficient middle ground between a dynamic call tree (DCT for short) and a dynamic call graph. CCT is defined by the following equivalence relation on a pair of vertices in a DCT. Vertices u and w in a DCT are equivalent if:

- v and w represent the same procedure, and
- the tree parent of v is equivalent to the tree parent of w, or $v = w$, or there is a vertex u, such that u represents the same procedure as v and w and u is an ancestor of both v and w.

The equivalence classes of vertices in a DCT define the vertex set of a CCT. Let $Eq(x)$ denote the equivalent class of vertex x. There is an edge $Eq(u) \to Eq(w)$ in the CCT iff there is an edge $u \to w$ in the DCT. A CCT compactly represents all calling contexts encountered during a particular program execution. The authors showed that both the breadth and the depth of the CCT in a program without recursion are bounded by the number of functions in a program and each individual vertex requires constant amount of memory.

As a performance improvement over the original idea of CCT, the parallel approach of CCT creation was proposed by Binder et al. [6]. The idea is to use a thread-local shadow stack [4, 16] of the current call context and the sequence (called packet) of functions calls and returns that resulted in the current call context. When that sequence reaches the predetermined length, it is pushed to the shared queue for further processing, then reset to the current call context as determined by the shadow stack. A separate working pool of multiple threads managed by *CCT manager* handles the queue and constructs the complete CCT asynchronously. Using a thread-local shadow stacks and asynchronous CCT constructing greatly reduced the impact on program runtime by reducing thread contention on the shared CCT representation and by maximising the utilisation of multiple cores of modern CPUs. The authors report the runtime overhead factors of 3.49 in the Dacapo benchmark suite (dacapo-2006-10-MR2).

Another technique for speeding up CCT creation was proposed by Huang and Bond [12]. The authors claim that previous CCT implementations have two major sources of overhead: the need to lookup child nodes while recording the transfer of control from caller to callee, and the creation of many CCT nodes that will not be used in a profiling session as no events of interest will ever happen in

	Intrusiveness	Precise	Works on HotSpot	Overhead ratio	Memory
CCT	low	+	can be transferred	1.7* / 2-50	$< 10MB$
Parallel CCT	low	+	+	3.49	hundreds MBs
CCU	high	+	-	1.3	3-50MBs
PCC	high/low	-	can be transferred	-	0
Breadcrumbs	high	+/-	-	2	tens GBs
PCCE	high	+	-	< 1.1*	0
DeltaPath	high	+	+	1.32	-
DACCE	low	+	-	< 1.08*	-

Table 1: Overview of different methods for obtaining calling context

functions represented by those nodes. To eliminate these problems, the authors propose using another data structure, *calling context uptree* (CCU for short), where each node, representing a function call in the program, points up to its parent in the current call context. This eliminates the child lookup overhead entirely, but it leads to the creation of new nodes on every function call as already existing nodes cannot be found and reused. The overhead of unneeded CCT nodes is solved by creating a CCU node representing the current call context as part of the recording of the profiling event metadata and recursively filling the CCU's reference to its parent by walking the stack until the function with an existing CCU is found. Then the newly created CCU is stored as part of the recorded event metadata.

Before presenting the summarized comparison of the above approaches, we would like to explain one of the criteria used, *intrusiveness*. Intrusiveness was described by Šor and Srirama [17] as "a special criterion, which describes how much effort is required in order to use the implementation of the proposed method". When some technique requires separate training runs of the application, the results of which have to be fed back to the tool before actual profiling execution, we consider this technique to be more intrusive. In large companies with long deployment cycles, this can be quite a prohibitive requirement. On the other end of the spectrum, there are tools implemented by simple Java agent using JVM TI standard API and bytecode instrumentation. In addition, every technique or tool, which requires using Jikes VM, as opposed to the standard HotSpot JVM, is considered being very intrusive.

The majority of the evaluated approaches for obtaining calling context turned out to be too intrusive. Only CCT [3], Parallel CCT [6] and DACCE [13] do not require neither modified JVM nor separate preprocessing step. PCC [8], although originally implemented on Jikes VM, seems to be quite easily transferable to HotSpot, thus lowering intrusiveness significantly.

Table 1 summarizes different approaches for providing calling context information to the applications' runtime profiling. As it turned out, there is only one tool which is not intrusive, works on HotSpot JVM and gives precise answers — Parallel CCT [6]. In comparison, our approach is admittedly somewhat slower, but it has the following advantages:

– Space overhead. In the case of the application without recursive calls Parallel CCT memory consumption is proportional to the square of the number of the functions in the application. If the application has deep recursive calls then memory overhead grows even more. Our approach stores only calling graph and thus requires memory proportional to the number of the functions.
– In order to use Parallel CCT for augmenting profiling metadata with location information, a client has to send profiling events to the CCT manager and to record all profiling data their. In our approach the identification of the calling context is available right at the place, where profiling event occurs, making it very easy for the clients to use.

8 Conclusions and future work

We have presented an encoding scheme for calling context encoding in object-oriented multi-threaded Java applications. This scheme is particularly suitable for recording extremely frequent events. The algorithm handles dynamic call graph changes due to JVM class loading and polymorphism in a natural and straightforward way. We provide the full and exact call context in which an event took place; that is, we produce a similar trace to the JVM itself would produce also when the call sites are impenetrable, such as calls to Java from native code. Unlike the previously proposed algorithms, we can deal with recursive calls without any special care. No prior knowledge of the application or source code preprocessing is required and the algorithm can be used as a plugin or in cooperation with other monitoring and troubleshooting tools.

The current implementation has fairly high overhead. In this form, we cannot apply it in a production setting. While there is room for optimization, we have tried most obvious ideas, such as caching of addition values and offloading work into parallel threads. Our performance is competitive with other approaches that can similarly handle the full spectrum of method calls available on the JVM without tweaking the runtime system itself, but the question is if one could apply the ideas from the faster approaches to speed up plain and simple function calls, while still maintaining the capacity to deal with more complicated cases.

Bibliography

[1] SPEC cpu2006, 2006. URL `http://www.spec.org/cpu2006/`.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44(2):399–417, Jan. 2005. ISSN 0018-8670.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97*, pages 85–96. ACM, 1997.

[4] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8.

[5] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29*, pages 46–57. IEEE Computer Society, 1996.

[6] W. Binder, D. Ansaloni, A. Villazn, and P. Moret. Parallelizing calling context profiling in virtual machines on multicores. In *PPPJ '09*, pages 111–120. ACM, 2009.

[7] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190. ACM, 2006.

[8] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA '07*, pages 97–112. ACM, 2007.

[9] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses. In *PLDI '10*, pages 13–24. ACM, 2010.

[10] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[11] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGOPS Oper. Syst. Rev.*, 38(5):156–164, Oct. 2004.

[12] J. Huang and M. D. Bond. Efficient context sensitivity for dynamic analyses via calling context uptrees and customized memory management. In *OOPSLA '13*, pages 53–72. ACM, 2013.

[13] J. Li, Z. Wang, C. Wu, W.-C. Hsu, and D. Xu. Dynamic and adaptive calling context encoding. In *CGO '14*, pages 120–131. ACM, 2014.

[14] Oracle. JVM Tool Interface, 2013. URL `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`.

[15] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *ICSE '10*, pages 525–534. ACM, 2010.

[16] A. Villazon, W. Binder, and P. Moret. Flexible calling context reification for aspect-oriented programming. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 63–74, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-442-3.

[17] V. Šor and S. N. Srirama. Memory leak detection in java: Taxonomy and classification of approaches. *Journal of Systems and Software*, DOI: 10.1016/j.jss.2014.06.005:139–151, 2014.

[18] V. Šor and S. N. Srirama. Memory leak detection in Plumbr. *Software: Practice and Experience*, 2014.

[19] Q. Zeng, J. Rhee, H. Zhang, N. Arora, G. Jiang, and P. Liu. DeltaPath: precise and scalable calling context encoding. In *CGO '14*, pages 109–119. ACM, 2014.