SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

May 2016, Vol. 59 052104:1–052104:15
doi: 10.1007/s11432-015-5418-3

# A new method to encode calling contexts with recursions

Lulu WANG[1], Bixin LI[1*] & Hareton LEUNG[2]

[1]*School of Computer Science and Engineering, Southeast University, Nanjing* 210096*, China;*
[2]*Department of Computing, Hong Kong Polytechnic University, Hong Kong* 999077*, China*

Received January 10, 2015; accepted June 8, 2015; published online April 12, 2016

**Abstract** Calling context encoding (CCE) uses some integers to represent the current context of any execution point, and is valuable in a wide range of applications. Existing studies already ensure accurate results of CCE, but they use stack to store the encoding when recursive calls happen. This may cost much for cyclic call graphs with deep recursive calls. This paper presents a novel approach (called RCCE) to address this problem. With a modified strategy for encoding, it requires less frequent access of the stack and gives more succinct representation of contexts on deep recursion. Experimental results show that compared to Precise Calling Context Encoding, RCCE is more practical and efficient on calling contexts with recursions.

**Keywords** calling context encoding, recursive calls, dynamic analysis, profiling, call graph

**Citation** Wang L L, Li B X, Leung H. A new method to encode calling contexts with recursions. Sci China Inf Sci, 2016, 59(5): 052104, doi: 10.1007/s11432-015-5418-3

## 1 Introduction

Control flow information of program execution is very important for behavior-based analysis, hence is very useful in lots of applications which try to observe and help improve program performance. For various uses, different aspects of dynamic control flow information are collected, such as paths (which show the sequence of executed statements), call paths (which show the sequence of methods/functions), and calling contexts (which show the context information at a callee method). Such aspects provide information at different granularity and require different overheads to collect on-the-fly. And in some applications, such information needs to be encoded into integer(s) for improved scalability.

Encoding helps to determine the frequency of program elements. This is because in certain cases we not only want to know what happened, but also how frequently it happened. For example in compiler optimization, it is necessary to collect frequency of paths. For the sake of counting how many times a path gets executed, it is practical to count the frequency of its "pathid" (an integer to represent the path), which would be far more efficient than counting the path itself (e.g., in the form of string).

In complicated cases with too many elements, it is not possible to accurately encode each element into just one integer. So they have to be encoded into a series of integers. Hence, shorter codes can generate frequency more easily, and lead to better performance.

* Corresponding author (email: bx.li@seu.edu.cn)

© Science China Press and Springer-Verlag Berlin Heidelberg 2016 

CCE (calling context encoding) [1] is used to encode the current calling context during an execution. Such an encoding is computed dynamically, and may be put to use at any execution position. This helps in software analysis and optimizations, and is useful in many context-sensitive approaches, such as monitoring, debugging, and event logging. The features of CCE also follow the general description above on control flow encoding.

CCE is similar to the path encoding in profiling since the calling context actually can represent paths in a call graph. But they use different types of encoding since profiling separates paths at the program exit while CCE separates contexts at a specific execution point. However, the basic methods of encoding used between them are quite similar.

PCCE (precise calling context encoding) [1, 2] can compactly encode acyclic calling contexts. It is based on a prior path profiling algorithm named EPP (efficient path profiling) [3], which encodes each acyclic path using unique encoding. PCCE uses EPP encoding to handle acyclic calling contexts, and by using a stack to store the encodings of acyclic fragments, it can precisely encode cyclic calling contexts of a cyclic call path.

Hence, PCCE may require too many accesses to the stack in the case of deep recursive calls. This will make PCCE inapplicable in some situations.

- Stack accessing is quite language dependent, i.e., it is very fast in C programs, and in object-oriented (such as Java) programs, and stack needs to be implemented with objects, so the pushing and poping of stack could be much slower. Therefore, many stack accesses may cost much running overhead for OO programs.
- The storage cost of PCCE is proportional to the recursion depth, so deep recursions may cause much storage cost and even stack overflowing error, and make the context encoding fail.
- For the usage of CCE, the encoding needs to be collected, counted and decoded, and such costs are proportional to the encoding length.

As recursive calls are very popular (as shown in our study, 76 packages out of 349 in JDK contain recursive calls, and many of them can involve large calling depths). To address this problem, we need to develop some new encoding approach that avoids pushing the encoding into the stack at each recursive call.

PAP (profiling all paths) [4] is an intra-procedural profiling approach, which precisely and efficiently encodes cyclic paths inside a procedure. This paper extends PAP into CCE to precisely encode each calling context.

In this paper we make two major contributions.

First, we extend PAP to RCCE (recursive calling context encoding) to encode calling contexts for deep recursive calls. PAP's encoding ability is used to handle the recursive calls in CCE, which results in much shorter encoding than PCCE with deep recursion; corresponding changes are made to turn PAP into RCCE, in order to meet the gap between path profiling and CCE.

Second, we evaluate RCCE against PCCE with experiments based on some benchmark programs to verify their relative strengths, where acyclic and recursive programs are separately used.

This paper is organized as follows: Section 2 presents the background concepts and key profiling approaches; Section 3 discusses the details of RCCE to encode calling contexts; Section 4 gives experimental evaluations; Section 5 summarizes related work and Section 6 concludes the paper.

## 2 Background

This section gives necessary background concepts and approaches. Subsection 2.1 illustrates the concepts of calling context in comparison to profiling; Subsection 2.2 reviews EPP, which forms the base for PCCE and PAP; Subsection 2.3 describes PCCE for encoding calling contexts, and Subsection 2.4 describes PAP for profiling intra-procedural flows and its breakpoint mechanism for supporting profiling of executed paths only.

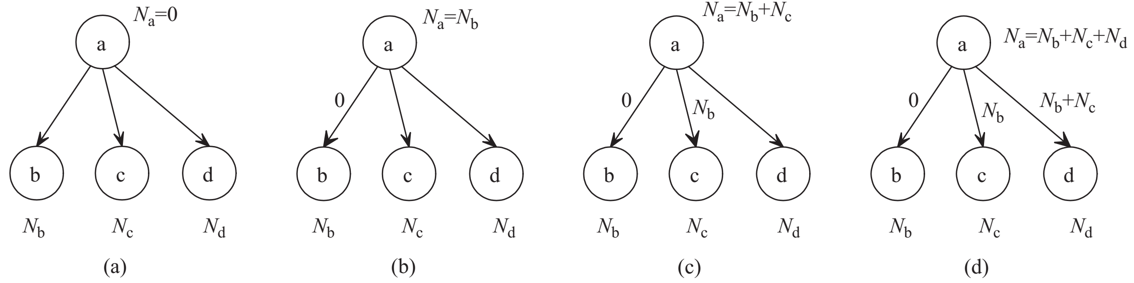


**Figure 1** Probes of EPP. (a) Father and children nodes; (b) deal with (a, b); (c) deal with (a, c); (d) deal with (a, d).

## 2.1 Motivation

Both CCE and (path) profiling encode dynamic control flows. At the procedure level, their encoding objectives are so called contexts and method sequence respectively. Note that there are key differences between calling context and method sequence. First, calling context differentiates multi-calls (one method calls another more than once, and such calls are shown as different call edges on the call graph) while method sequence does not. Second, method sequence differentiates calls with different returns while calling context does not.

Therefore, in calling context encoding, if the encoding changes when a method call occurs (for example, the encoding adds an integer), it needs to be restored after the call returns (that is, it subtracts the integer).

Because of the differences of their encoding targets, CCE cannot use the encoding from profiling directly; some modifications are needed for both PCCE [1] and RCCE in this paper. PCCE modifies the encoding of EPP which only encodes acyclic paths, and uses stack to turn a cyclic context into a series of acyclic fragments in order to ensure a precise encoding for recursive calls.

Different from EPP, PAP precisely encodes both acyclic and cyclic paths. So benefiting from this, RCCE does not access the encoding stack at each recursive call.

## 2.2 Efficient path profiling (EPP)

In order to collect path profiles, existing approaches use probe variable(s), that is, they instrument probe statements into the target program, and these statements are executed as the target program runs. After execution, the value of the variable(s) can be used to determine the specific path of execution. So profiling contains three steps: initialization (give default encoding to the probe variable(s)), computation (change the value of variable(s) as program runs), and collection (store the encoding results of the paths, and get the path frequency with other execution). This profiling process requires different values assigned to different paths to ensure correct and accurate profiles.

EPP uses an integer variable $r$, and its probes are in the form of "$r := r + i$", where "$i$" is the weight of the edge, which is computed using Ball-Larus algorithm [3]. Figure 1 shows how EPP computes the edge weights in a DAG (directed acyclic graph), where "$N_{\rm a}$" denotes the NoP (number of paths) from node "a". The NoP from the father node is the sum of the NoP from all its children. So EPP computes weights (which determines the probes) on father-children edges one by one, as Figure 1(b)$\sim$(d) show, the weight of the current edge is equal to the current NoP of the father node, and then the NoP of the child adds up to NoP of the father. In this way, EPP gives each acyclic path a unique pathid [5], so that the probe value of variant "$r$" can be used to identify the corresponding path after execution.

Figure 2 shows how EPP instruments acyclic and cyclic CFGs (the statements underlined are instrumentation). Figure 2(a) is an acyclic program, and in Figure 2(b) EPP encodes the three paths into 0, 1 and 2. Figure 2(f) is a cyclic program, and EPP does not encode cyclic paths: whenever a backedge is executed, the encoding is ended (sometimes resumed in order to profile acyclic path fragments).

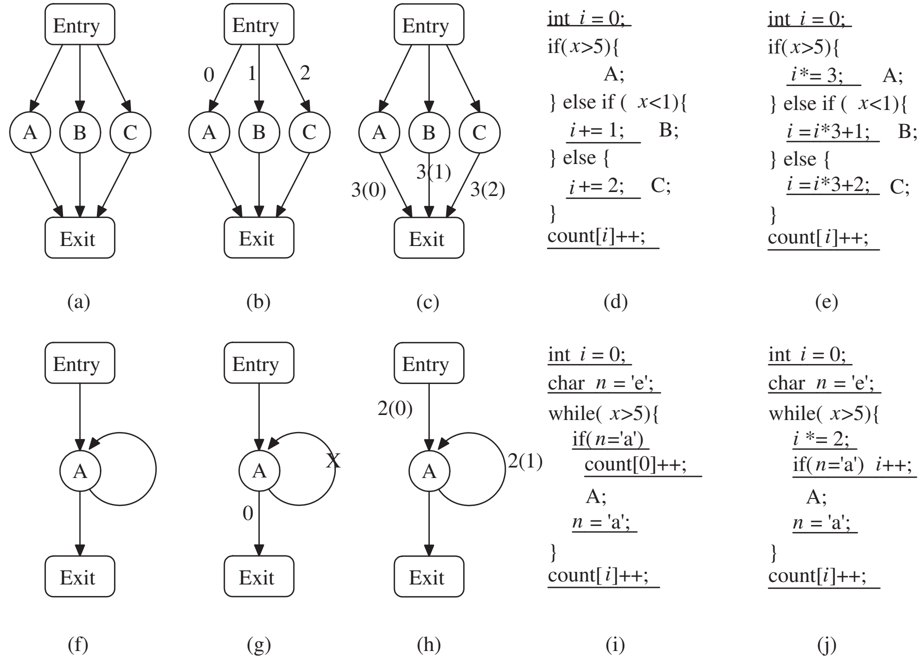


**Figure 2** A profiling example of EPP and PAP. (a) Acyclic CFG; (b) instrumented by EPP; (c) instrumented by PAP; (d) implementation of EPP; (e) implementation of PAP; (f) cyclic CFG; (g) instrumented by EPP; (h) instrumented by PAP; (i) implementation of EPP; (j) implementation of PAP.

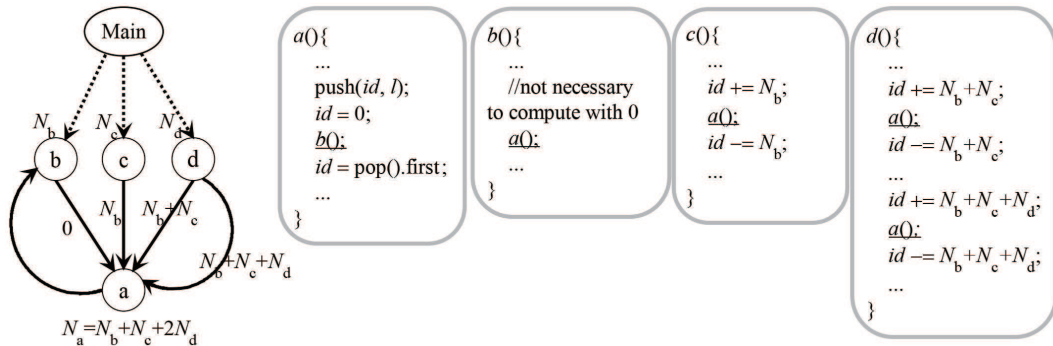


**Figure 3** An example of PCCE algorithm and instrumentation.

### 2.3 Precise calling context encoding (PCCE)

PCCE [1, 2] is designed to uniquely represent the current context of any execution point using a small number of integer identifiers. The basic PCCE algorithm of weight assignment (Algorithms 1 & 3 in [1]) and instrumentation can be described in Figure 3. Similar to EPP, PCCE assigns weights to call edges of an acyclic call graph so as to ensure an unique encoding for each calling context; and for the backedges (recursive calls), it uses a stack to push and pop the encoding when calls are made and returned. When a calling context is required, the encoding is decoded into the original context (Algorithms 2 & 3 in [1]).

Thus, it can be inferred that when there are many recursive calls in a call chain, the stack may become very deep and inefficient. In this paper, we develop a context encoding approach to address this problem of large stack requirement. Our innovation is to use the stack only when the encoding overflows and not when a backedge executes.

### 2.4 PAP and its breakpoint mechanism

In order to profile all paths (especially cyclic paths), we have developed the PAP algorithm [4], which instruments probes in the form of "$r := r * s + k$" ($s(k)$ for short, "$s$" is called weight and "$k$" is called remainder) on the inedges of each CFG node N, where $s$ is the indegree of N, and $k$ is the serial number of the inedge, no matter acyclic or cyclic, as shown in Figure 2(c) and Figure 2(h). After execution, using the probe value of $r$, we can determine the whole path by iteratively finding out the former node and restoring its probe value. PAP can accurately profile both acyclic and cyclic paths with a slightly larger number of probes [4].

PAP uses multiplication and addition in probe computing. For a given cyclic path, it may contain many iteration of loops so that the probe value may grow very big and overflow. Therefore, a breakpoint mechanism is introduced to handle the probe-value overflowing that may happen in long paths: when the probe value overflows after executing some edge (M, N), M and the probe value at M are stored as a breakpoint. The probe value at N is set to be the remainder of the probe on edge (M, N). Each breakpoint determines a path part, and all the parts can be connected together to determine the whole path [4].

Compared to other profiling approaches [3, 5–10], instead of ignoring long paths, the breakpoint mechanism gives the ability for profiling long paths with extra cost only for those paths that are actually executed. Given the above consideration, the storage cost of breakpoints should be reasonable.

## 3 Recursive calling context encoding

In this section, we propose a new approach for encoding calling contexts, called RCCE, which overcomes the large stack problem of PCCE by using a different encoding. RCCE is designed to encode calling contexts with recursive calls so that it does not require as many stack accesses as that of PCCE in order to improve efficiency in execution and context decoding. Similar to PAP, RCCE assigns weights to call edges, and computes the encoding dynamically. The probes used by RCCE follow the same form as PAP probe, i.e., addition and multiplication are both used. Also, a stack is used when overflowing happens during encoding computation. Different from PAP, RCCE only deals with call edges, not return edges, and the encoding is restored to its previous value before a call after the callee method returns. However, decoding of RCCE may be necessary at each calling context, while that of PAP is only needed when a path terminates. Different from PCCE, RCCE uses multiplication in encoding and the stack is used when overflowing happens, and not for each recursive call.

Briefly, the relationship among EPP, PAP, PCCE and RCCE can be described as (1) PAP enhances the encoding ability of EPP; (2) PCCE uses EPP encoding for CCE with a stack to deal with recursive calls and overflowing; (3) RCCE uses PAP encoding for CCE with a stack to deal with overflowing.

Next we discuss the details of the instrumentation, encoding and decoding of RCCE.

### 3.1 Instrumentation and encoding

Figure 4 shows how RCCE works on the call graph shown earlier in Figure 3. RCCE uses different assignments of different types of probes from PCCE. Figure 4 uses the instrumentation of $c()$ as an example. Since the probe "4(1)" is assigned to call edge (c, a), it is correspondingly instrumented next to the call statement $a()$ (underscored); as given to Figure 4(b) which displays the source code of $c()$ calls $a()$ and the code instrumentation. The instrumentation in Figure 4(b) (named as "RCCE-") can not handle overflowing of encoding, and to fix that, we use the breakpoint mechanism of PAP for the encoding, which results in Figure 4(c) (named as "RCCE").

Such an instrumentation of RCCE can ensure correct calling context encoding because:

• The variable "*id*" is initialized to integer "1" at the very beginning, and if overflowing happens, its value is pushed into the stack and reset to 1, so that the encoding is precisely stored;

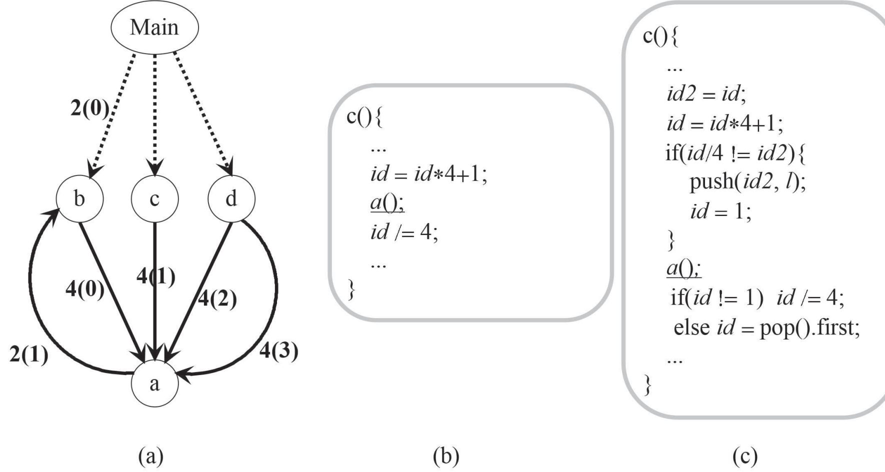


**Figure 4** An example of RCCE instrumentation. (a) Probes on flow graph; (b) instrumentation of RCCE-; (c) instrumentation of RCCE.

- After the computation $id$ /= 4, $id$ is restored to its value before $id = id * 4+1$, so that it precisely encodes the contexts;
- After the callee method $a()$ returns, if the value of $id$ is not 1, we can know that its value was not pushed and reset before $a()$ was called;
- After the callee method $a()$ returns, if the value of $id$ is 1, we can know that its value was pushed before $a()$ was called since before the probe statement, $id$ was at least 1. Thus, no matter what probe statement $id = id * s +$ i is executed, $id$ is definitely not 1. Accordingly, after the method call returns, $id$ is correctly restored to its previous value before the method call.

With such an instrumentation, RCCE precisely encodes each calling context (same as PCCE), and avoids pushing the current encoding into the stack at each recursive call (as well as the required popping when the call returns).

### 3.2 Decoding

During the execution, when the current context is required, the context encoding is transferred into the corresponding explicit context.

In PCCE, the encoding is used to calculate the previous call edge and this process is repeated iteratively until it gets to the entry node, as shown in Algorithm 2 of [1], which means that at any position (named as X) of execution, the currently computed encoding can precisely determine the call edge (named as Y) before X, and then the encoding before Y can be found out from the instrumented edge weights.

Similar to PCCE, RCCE uses the current encoding to compute the explicit context with the instrumented edge weights; different from PCCE, RCCE uses multiplication besides addition in encoding, so its decoding uses division instead of subtraction as the reverse computation of encoding.

### 3.3 A simple example for comparison

Here we use a simple program to show the benefit of RCCE over PCCE. The recursive program implements the function of computing the factorial of a given integer.

As shown in Figure 5(a), the method main() is inputed an integer array and calls method $a()$ to compute their factorials, while method $a()$ uses recursion to achieve this. For RCCE, as shown in Figure 5(b), the two edges of the call graph (one is backedge) are instrumented to separate the encodings of different contexts. And for PCCE, as shown in Figure 5(c), it uses a dummy node to deal with the backedge (e.g., turn a backedge into an edge from the dummy node), and uses the stack operation to encode contexts with the backedge executed. Some sample contexts and encodings are also shown in

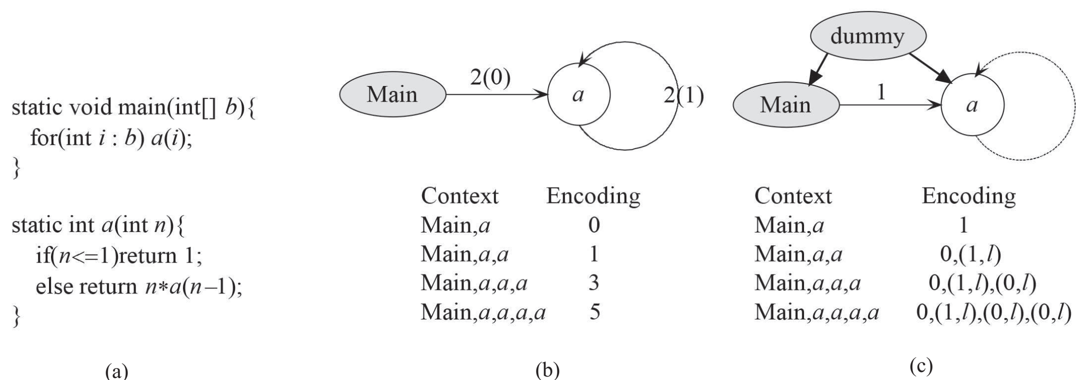


**Figure 5** A simple program for RCCE. (a) the program; (b) RCCE is performed; (c) PCCE is performed.

**Table 1** Acyclic benchmark programs of JDK

| Package | Entry method | Call edges |
|---|---|---|
| com.sun.imageio.plugins.gif | GIFImageWriter:computeRegions | 9 |
| com.sun.imageio.plugins.common | ImageUtil:getTileSize | 13 |
| | ImageUtil:getBandSize | 7 |
| com.sun.imageio.plugins.jpeg | AdobeMarkerSegment:writeAdobeSegment | 12 |

Figure 5(b) and (c), which demonstrate that RCCE avoids stack operation on each backedge execution and gives more concise encodings on recursive calls.

## 4 Experiment

This section presents an empirical evaluation of the performance (efficiency) of RCCE when compared to PCCE.

As discussed earlier, the motivation of developing RCCE is to overcome the problem of PCCE which pushes too many encoding into the stack for recursive method calls. Thus, on one hand, we would like to find out how RCCE performs on acyclic calls, and on the other hand, we would like to evaluate its benefit on recursive calls.

### 4.1 Experimental setup

#### 4.1.1 *Metrics*

In calling context encoding, the overhead can be classified into three categories: the time cost of execution after instrumentation, the space cost of storing the encoding in the stack, and the time cost of decoding the context at selected positions. The time cost can be measured by the actual time, while the space cost can be measured by the maximum stack size during execution, and the decoding cost can also be measured by the stack size at decoding positions during execution (because it may be necessary to get the original contexts anywhere during execution, we use an independent thread to access the encoding during the whole execution, and use the average as the metric of decoding cost).

#### 4.1.2 *Benchmark*

We like to compare the performance of RCCE and PCCE on acyclic calls first and then recursive calls. We look up JDK (Java development kit) package, which is a development environment for the Java TM platform that includes various Java programs [1)], to find proper benchmark programs.

On acyclic call graphs, we select four programs from all JDK programs with static entry method and most call edges, which are listed in Table 1, including the number of call edges of their call graphs.

1) JDK (Java Development Kit) 6.0, http://www.oracle.com/technetwork/java/index.html.

**Table 2** Recursive benchmark programs of JDK

| Package | Entry method | Explicit recursive calls |
|---|---|---|
| java.util.regex | Pattern:matcher | 13 |
| java.util | Formatter$FormatSpecifier:print | 21 |
| com.sun.org.apache.xerces.internal.impl.xs.traversers | XSDHandler:parseSchema | 30 |
| com.sun.org.apache.xerces.internal.impl.xpath.regex | RegularExpression:matches | 47 |

On recursive call graphs, we use the JDK programs with most explicit recursive calls, which are listed in Table 2 [2)].

So in all, we use eight programs (renamed as Adobe, Band, Gif, Tile, Regex, Util, Xs and Xpath for short) to test the performance of PCCE and RCCE.

Other than JDK, we also use two popular Java benchmark systems: DaCapo[3)] and SPECjvm2008[4)] (the former contains recursion and the latter does not). They both consist of many benchmarks and are used in Subsection 4.5.

#### 4.1.3 *Implementation*

Our experiment is done in the following environment: Windows XP SP3 with Eclipse 3.6; the CPU is Intel Q6600 (2.40 GHz) with 2G memory.

We have implemented PCCE and RCCE in Java, based on ASTParser (a tool which extracts the abstract syntax tree of Java source code, from JDT, Eclipse 3.6) for our experiments. Besides, we use the following plug-ins: org.eclipse.jdt.core and org.eclipse.equinox.common, which are used for parsing source code of benchmark programs and creating the call graphs; org.eclipse.text, which is used to record our instrumentation and apply it back into the source code files; jxl, which exports experimental results into an excel file. Other jars are used only if they are imported by the benchmark programs.

Eclipse is running with default settings (-Xms40m, -Xmx384m).

### 4.2 Experimental results with random inputs

For each program, we first generate random parameters from its input domain, and then execute the program. This is repeated 100 times. The results are reported in Figures 6 and 7 respectively. Since the call depth is dependent on the selected input. The end result is that several inputs may invoke the same call depth, while others may invoke different call depths. Thus, the number of data points for different programs are not the same.

From Figure 6, we can see that for acyclic call graphs (Adobe, Band, Gif and Tile), the storage cost and decoding cost of PCCE and RCCE are the same since no stack-pushing is needed; and from Figure 7, we can see that for cyclic call graphs (Regex, Util, Xs and Xpath), the storage and decoding costs of PCCE are obviously more than RCCE.

On the time cost, most data overlap, which means the time costs of PCCE and RCCE on the executions with the same call depth are mostly the same. So we use the average value over 100 executions to evaluate this cost, as reported in Figure 8(a)("orig" means the time cost of the original programs without instrumentation of PCCE or RCCE), while the average storage and decoding costs (for Regex, Util, Xs and Xpath) are shown in Figure 8(b) and Figure 8(c) respectively. Note that the vertical axes of all three figures are in logarithmic scale.

Figure 8(a) shows that on most programs (6 of 8, except Tile and Util) PCCE and RCCE have similar time cost (compared to PCCE, the relative differences of RCCE are 0.95%, −3.44%, −4.64%, 9.93%, 0.05% and −3.64% on Adobe, Band, Gif, Regex, Xs and Xpath respectively), but for Tile and Util, RCCE costs more time than PCCE (111.59% and 130.91% respectively). This is caused by the extra cost for performing multiplication in RCCE encoding.

2) All JDK packages are inspected except six: java.lang.annotation, java.awt, com.sun.org.apache.regexp.internal, javax.swing.text.html.parser, avax.swing.plaf.basic, javax.swing, which are too complicated to analyze.

3) Dacapo-9.12-bach, the second major release, http://www.dacapobench.org/.

4) SPECjvm2008, http://www.spec.org/jvm2008/.

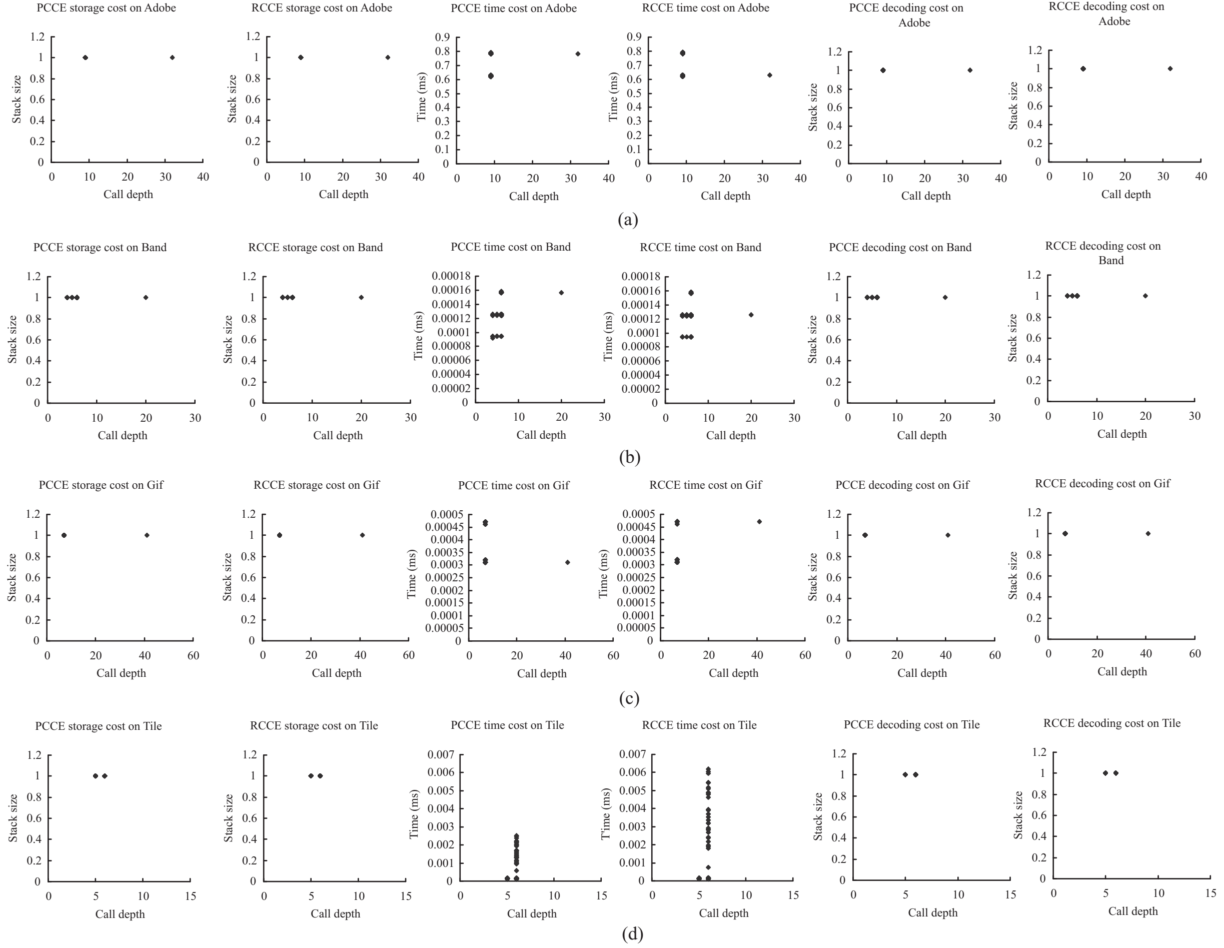


**Figure 6** Experimental results against call depths on acyclic calls. (a) Results on Adobe; (b) results on Band; (c) results on Gif; (d) results on Tile.

Figure 8(b) and (c) obviously show that PCCE uses a deep stack of encoding to represent a recursive context for Xs and Xpath, and RCCE uses a much smaller stack.

### 4.3 Analysis

From the results in Figures 6–8, we can draw the following observations: (1) for storage cost and decoding cost, RCCE and PCCE are the same on acyclic graphs, and PCCE costs more than RCCE on recursive call graphs; (2) for time cost, RCCE and PCCE are approximately the same on most programs, and RCCE costs more (about twice) than PCCE for a small number of programs; (3) RCCE is more efficient than PCCE on storage and decoding costs as the call depth grows, but may require more time on some light recursive calls. Thus, RCCE should be used on programs with deep recursive calls.

Here we would like to know the depth of recursion that RCCE is most suitable for, and to get the boundary of using RCCE instead of PCCE. Since RCCE is already proved to be more efficient on storage and decoding costs than PCCE for recursive programs, we only need to observe the last type of cost: running overhead. If PCCE costs less, then it is applicable when users value running overhead much; if PCCE costs similar or even more than RCCE, then RCCE should be applicable. The programs with deepest recursion, Xs and Xpath, are used for this comparison, and the results are shown in Figure 9, which indicates that:

- On Xs, with recursion depth of [0, 5), RCCE costs more than PCCE; with depth of [5, 178), RCCE costs obviously less than PCCE; while with depth bigger than 178, they cost almost the same. So RCCE is applicable with recursion depth bigger than 5.

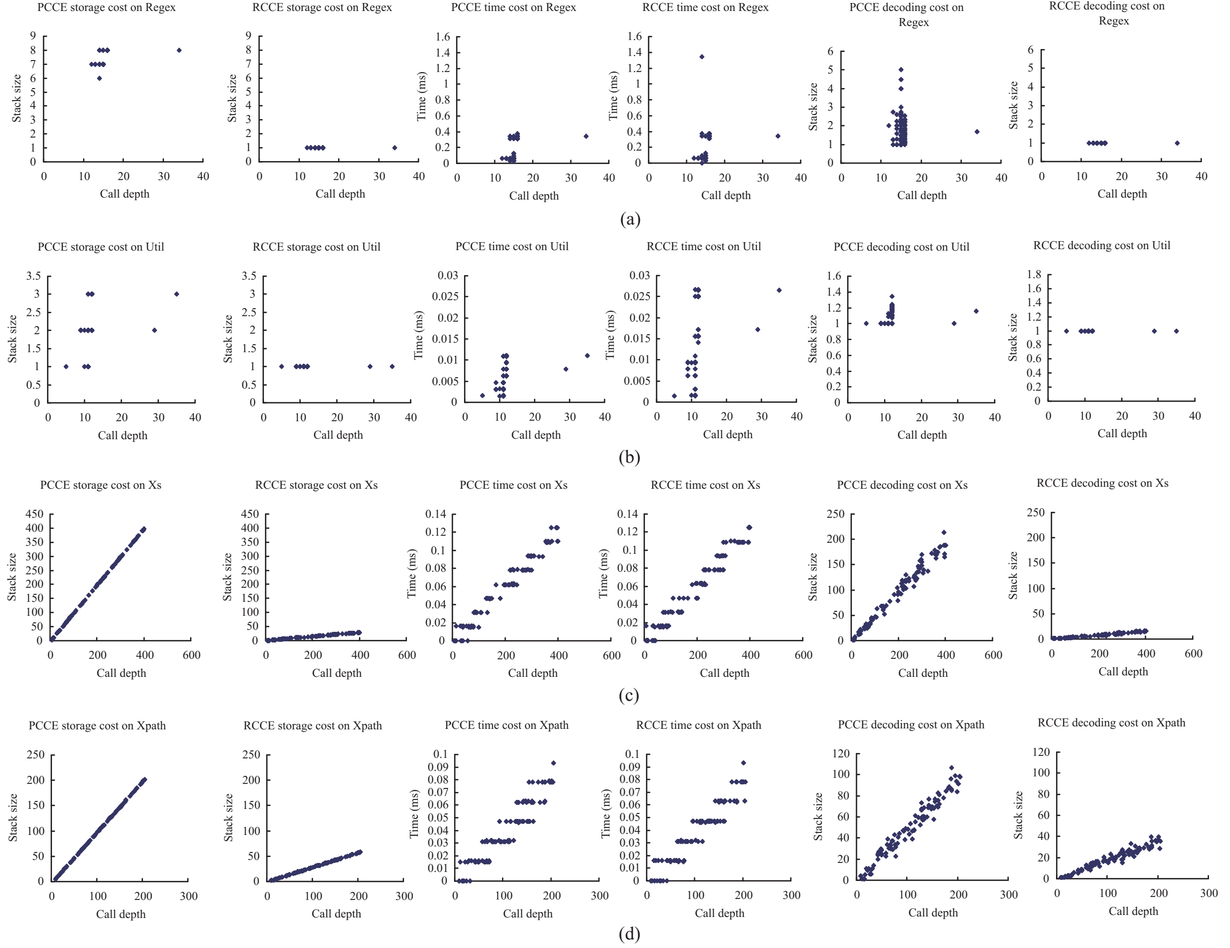


**Figure 7** Experimental results against call depths on recursive calls. Results on (a) Regex, (b) Util, (c) Xs, and (d) Xpath.

• On Xpath, with recursion depth of [0, 54), RCCE costs less than PCCE; with depth bigger than 54, they cost almost the same. So RCCE is applicable on all recursion depths.

From the results in Figure 9, it can be inferred that:

• There are two boundaries for recursion depth, treated as $B1$ and $B2$, which divide the depth into three intervals: [0, $B1$), [$B1$, $B2$), [$B2$, max depth].

• PCCE costs less than RCCE in [0, $B1$); PCCE costs more than PCCE in [$B1, B2$); they cost similar in [$B2$, max depth].

• PCCE is applicable in [0, $B1$), and RCCE is applicable in [$B1$, max depth].

• The values of $B1$ and $B2$ vary from program to program. When $B1$ equal to 0, RCCE is applicable to all recursion depths.

### 4.4 Experimental conclusion and threats analysis

From our experiments, we can draw the following conclusions:

• For calling context encoding of deep recursive calls, RCCE is more effective than PCCE as it has high efficiency in time cost, storage cost and decoding cost.

• For calling context encoding of acyclic calls or light recursive calls, RCCE performs similarly to PCCE and occasionally worse than PCCE; so PCCE is recommended for such cases.

We next discuss the most important threats to internal and external validity of our study. There are three possible threats to internal validity:

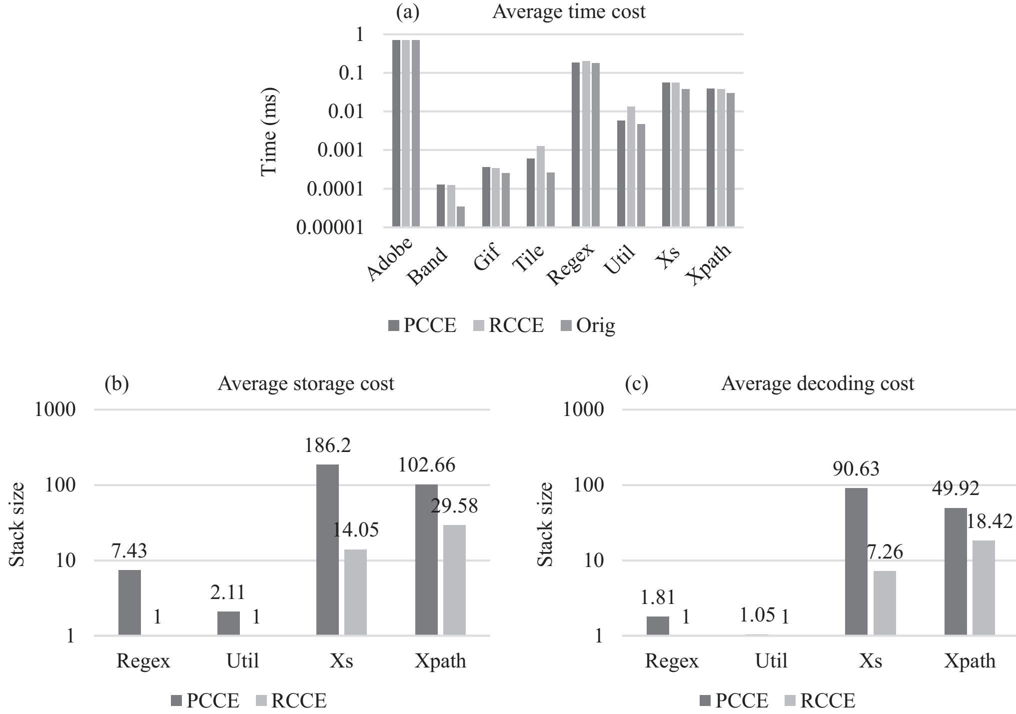


**Figure 8** Average costs of PCCE and RCCE. (a) Time costs; (b) storage costs; (c) decoding costs.

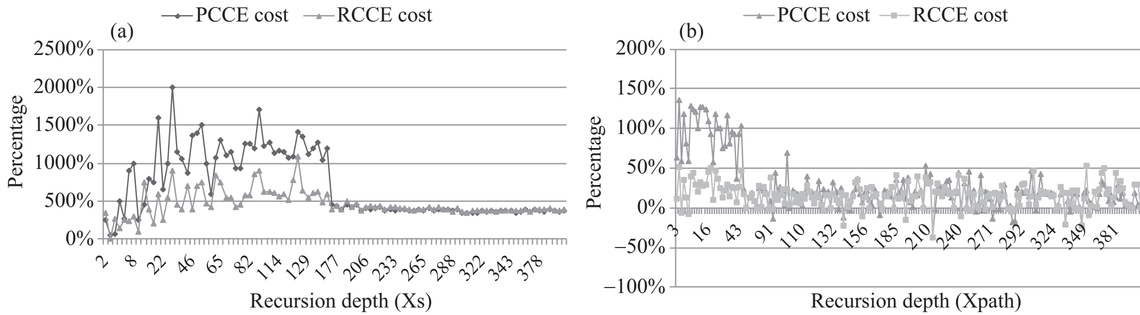


**Figure 9** Running overhead with recursion depth on (a) Xs, and (b) Xpath.

• Experimenter bias. Since the implementation automatically instrument programs, and the results are automatically reported, the bias of the experimenters should not affect the outcomes.

• Selection bias. The benchmark programs consist of two groups: acyclic and recursive. We automatically choose an initial set of programs and then manually choose the usable ones as the final chosen benchmark. This selection process is reasonable since we do not want to select programs with difficulty to set up running environment. The selection bias cannot be avoided.

• Accuracy of instrumentation. The experiments are done on the platform of Eclipse with JVM, which have high reliability. We have also checked the correctness of the instrumentation manually for several simple cases.

We identified two possible threats to external validity:

• Reliability of conclusions. Since eight programs of different types and various call depths are used, the conclusions should be able to cover programs of many similar structures.

• Representation of selected subjects. The selected benchmarks are in fact very few since we only choose from the JDK programs, and they may not be representative of many different types of programs. More wide-ranging experiments should be conducted to fully evaluate our proposed methods in the future.

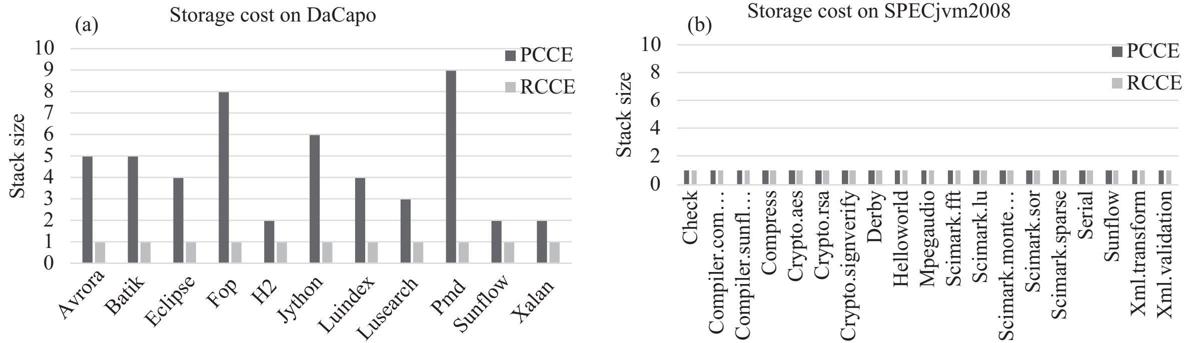


**Figure 10** Storage cost on (a) DaCapo programs, and (b) SPECjvm2008 programs.

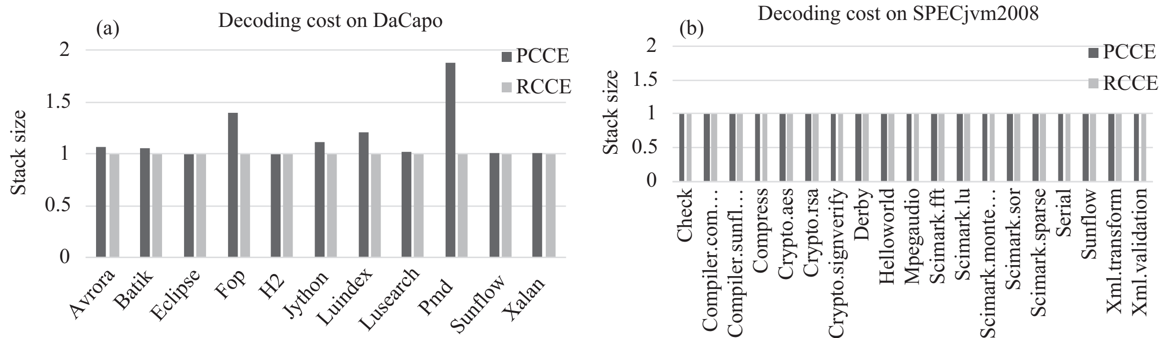


**Figure 11** Average decoding on (a) DaCapo programs , and (b) SPECjvm2008 programs.

## 4.5 Further exploration

In previous experiments, we have evaluated RCCE in comparison of PCCE on both acyclic and cyclic benchmarks, where we can observe how they perform with various call depths.

Here we inspect their performances on more complicated programs, using DaCapo, which consists of a set of open source, real world applications with non-trivial memory loads. The 11 programs of DaCapo[5) and all 19 programs of SPECjvm2008 (as listed in Figure 10) are instrumented by PCCE and RCCE, and then executed.

First we report the storage and decoding costs in Figures 10 and 11, in the same way of Figure 8(b) and (c).

From the results, we can see that on all benchmarks in this section, RCCE does not involve any overflowing. In the meanwhile, PCCE requires 1∼8 times more storage for storing encodings, and 0.01% – 87.3% more decoding costs.

Based on above results, we can see that no overflowing ever happens in RCCE execution, so we can use "RCCE-" here (aforementioned in Figure 4(b)), to do the same work with RCCE.

Four versions of benchmarks (the original, instrumented by PCCE, RCCE and RCCE-) are executed for their comparison of time costs, which is reported as the increase ratio after three types of instrumentation, as shown in Figure 12, which reports that: PCCE instrumentation increases the running time of DaCapo benchmarks by −6.4%–0.5%, −2.1% on average, and that of SPECjvm2008 by −5.0%–9.7%, 1.7% on average; RCCE increases that by −4.4%–14.1%, 4.3% on average, and that of SPECjvm2008 by −3.6%–10.3%, 1.6% on average; RCCE- increases that by −2.1%–11.8%, 2.0% on average, and that of SPECjvm2008 by −1.0%–8.7%, 1.9% on average.

Figure 12 indicates that: on DaCapo, RCCE mainly has more execution cost than PCCE (not always, except H2), and RCCE- effectively reduces that cost on RCCE (except H2, Jython and Luindex), but

5) In all DaCapo has 14 programs, where 3 of them (Tomcat, Tradebeans and Tradesoap) are not used in our experiment, because we need to execute one program iteratively to get more precise results, but these three cannot do this on our platform.

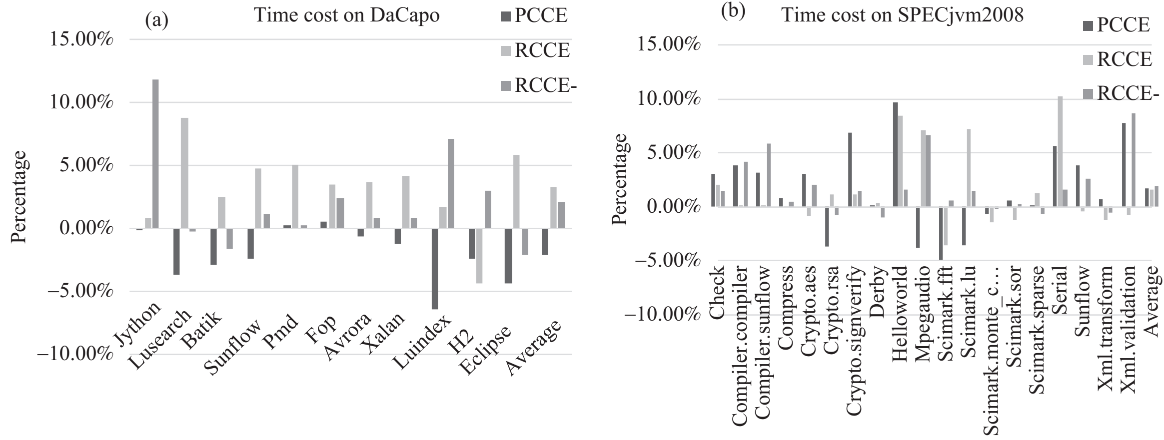


**Figure 12** Average time cost on (a) DaCapo programs, and (b) SPECjvm2008 programs.

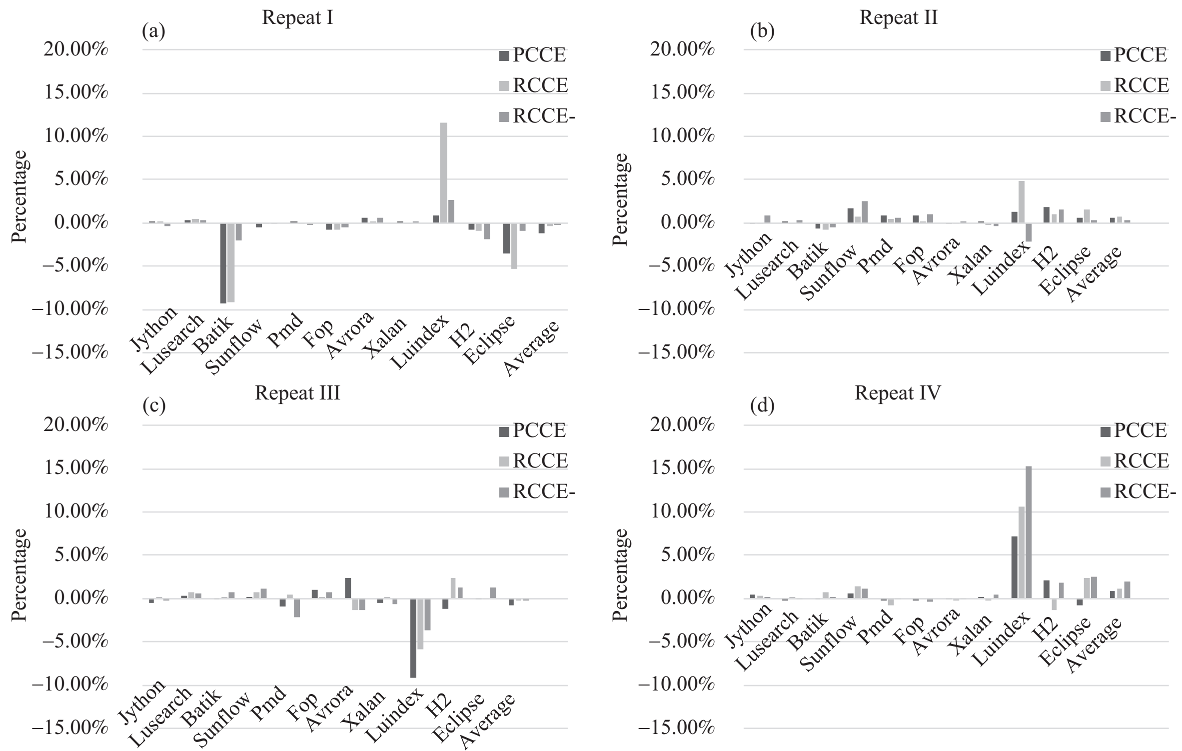


**Figure 13** Average time cost on DaCapo programs. (a) Repeat I; (b) Repeat II; (c) Repeat III; (d) Repeat IV.

still costs more than PCCE (except Avrora); on SPECjvm2008, the three do not have much difference and the benchmark does not involve any recursions.

Figure 12 gives totally different results from our previous experiments: sometimes the running time of a program gets (a little) less after instrumentation. For example, on batik, PCCE instrumentation reduces running time by 2.9%, while RCCE- reduces that by 1.6%.

This result should be due to two types of causes: internal causes of the program (possibly the compiler optimization on the instrumented code, better load balance on the multi-cores, etc.) and the external cause, i.e., the noise. The internal causes are acceptable since they represent the behavior of the program, while the external cause means that our experiment reports improper results. We need to inspect if the result (instrumentations reduce the running overhead) is caused by the external cause.

In order to study the noise in our execution, we repeat the experimental phases four times to see if the results are similar. The results are shown in Figure 13, which indicates that noise does exist in our

experiments since the results are different, but it should not be the reason for instrumentations reducing the overhead since same observation is obtained from every repeated running.

In this way, it makes clear that the noise can be basically eliminated, and the results reported in Figure 12 should be reasonable.

In order to reduce the effect of noise, we use the average data of all repeated experiments: PCCE has the overhead of −0.55%, RCCE has the overhead of 0.91%, and RCCE- has the overhead of 0.79%.

Therefore, such results should really exist, and could be caused by compilation optimizations, since instrumented codes may be compiled in a different way from the original codes.

## 5 Related work

The calling context was previously defined as the chain of method calls (i.e., the sequence of un- returned method calls from the program's root method to the method invocation [11]) that are concurrently active on the stack [12]. CCE refines this definition since it differentiates multi-calls. So naturally, it is logical to access the call stack to get the contexts, and Stack Walking is a popular calling context encoding, which traverses the program stack whenever a context is requested in order to construct the full identifier for the context [13]. The disadvantage of this method is that the calling stack may not be easy to access for some cases.

Another way for encoding context is to use the CT (Call Tree) or CCT (Calling Context Tree) [12, 14, 15]. CTs are built dynamically, i.e., for certain executions. When a method invocation occurs, the corresponding edge is added to the CT, and meanwhile, different nodes of the same method may be used to make sure that there are no backedges (recursive calls) on the tree. CT shows different method invocations of the same nodes while CCT merges them, i.e., CCT is a simple form of CT. Each path on the tree (CT or CCT) represents a context and can be encoded into an integer. But such trees need to be maintained dynamically and it is not as succinct as the encoding of PCCE [1].

Similar to selective path profiling [5, 6] which handles only a subset of paths for better performance, ICPP (incremental call-path profiling) encodes the contexts of a subset of all functions in a program [16]. This approach instruments only the functions of interest to the user, and uses an inexpensive stack walk mechanism to record the call-path to a function and work out the performance data. In this way, total overhead is reduced while only the interesting paths are dealt with.

Unlike PCCE and RCCE, some approaches produce a "probabilistically" unique id for each calling context [17]. The accuracy of result may be traded-off for better efficiency. For example, Ref. [18] uses the height of the call stack to identify the currently executing function, which is "probabilistically" unique at a low cost.

Recently, some research focuses on CCE and tries to make improvements based on PCCE. Ref. [19] presents DeltaPath for both procedural and objected-oriented programs, and its efficiency is evaluated as comparable with PCCE. DACCE (dynamic and adaptive calling context encoding) [20] focuses on indirect calls in C programs, and uses dynamic encoding (encodes the function call edges invoked at runtime) for better efficiency. DACCE costs less than PCCE at runtime overhead, conversely, our work focuses on recursive calls in OO programs and uses smaller stack to encode than PCCE.

## 6 Conclusion

It is well known that PCCE [1] can give precise and compact encoding on contexts. But since it is based on EPP algorithm which lacks the ability of dealing with recursions, PCCE uses stack operations to handle recursive calls (which may lead to encoding overflow). This may lead to a high cost for deep recursive calls.

RCCE is introduced to address this problem by accessing the stack only when the encoding overflows. RCCE is an extension of PAP algorithm to encode calling contexts. But since RCCE uses more complicated probes for encoding and extra computation for dealing with stack overflow, its efficiency is lower

than PCCE on acyclic calls.

Our experimental results show that RCCE is better than PCCE on storage and decoding costs for recursive programs, but not better or even worse than PCCE on several acyclic or light recursive calls. So the application of RCCE should be related to the specific programs.

Our experiment uses eight JDK benchmark programs (four acyclic and four recursive) to test the performance of RCCE with various call depths; 11 DaCapo and 19 SPECjvm2008 benchmark programs with default inputs. More wide-ranging evaluations are still necessary to test this method under other situations. Furthermore, as [1] proposes hybrid encoding to improve the encoding efficiency by simplifying the instrumentation from the information provided by the stack offsets, it may also be valuable to apply the same idea to RCCE.

**Acknowledgements** This work was supported partially by National Natural Science Foundation of China (Grant No. 61402103), and Jiangsu Natural Science Foundation (Grant Nos. BK20130633, BK20140644).

**Conflict of interest** The authors declare that they have no conflict of interest.

## References

1 Sumner W N, Zheng Y, Weeratunge D, et al. Precise calling context encoding. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, 2010. 525–534
2 Sumner W N, Zheng Y, Weeratunge D, et al. Precise calling context encoding. IEEE Trans Softw Eng, 2012, 38: 1160–1177
3 Ball T, Larus J R. Efficient path profiling. In: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture (MICRO), Paris, 1996. 46–57
4 Li B, Wang L, Leung H, et al. Profiling all paths: a new profiling technique for both cyclic and acyclic paths. J Syst Softw, 2012, 85: 1558–1576
5 Vaswani K, Nori A V, Chilimbi T M. Preferential path profiling: compactly numbering interesting paths. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Nice, 2007. 351–362
6 Apiwattanapong T, Harrold M J. Selective path profiling. In: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE), Charleston, 2002. 35–42
7 Bond M D, McKinley K S. Practical path profiling for dynamic optimizers. In: Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Jose, 2005. 205–216
8 Tallam S, Zhang X, Gupta R. Extending path profiling across loop backedges and procedure boundaries. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Jose, 2004. 251–264
9 Roy S, Srikant Y N. Profiling k-iteration paths: a generalization of the Ball-Larus profiling algorithm. In: Proceedings of the 7th International Symposium on Code Generation and Optimization (CGO), Seattle, 2009. 70–80
10 Joshi R, Bond M D, Zilles C. Targeted path profiling: lower overhead path profiling for staged dynamic optimization systems. In: Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO), San Jose, 2004. 239–250
11 Zhuang X, Serrano M J, Cain H W. Accurate, efficient, and adaptive calling context profiling. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Ottawa, 2006. 263–271
12 Ammons G, Ball T, Larus J R. Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, 1997. 85–96
13 Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, 2007. 89–100
14 Spivey J M. Fast, accurate call graph profiling. Softw Pract Exper, 2004, 34: 249–264
15 Villazon A, Binder W, Moret P. Flexible calling context reification for aspect-oriented programming. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD), Charlottesville, 2009. 63–74
16 Bernat A R, Miller B P. Incremental call-path profiling. Concurr Comput Pract Exper, 2007, 19: 1533–1547
17 Bond M D, McKinley K S. Probabilistic calling context. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Montreal, 2007. 97–111
18 Mytkowicz T, Coughlin D, Diwan A. Inferred call path profiling. In: Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Orlando, 2009. 175–189
19 Zeng Q, Rhee J, Zhang H, et al. DeltaPath: precise and scalable calling context encoding. In: Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Orlando, 2014. 109–119
20 Li J J, Wang Z J, Wu C G, et al. Dynamic and adaptive calling context encoding. In: Proceedings of the 12th IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Orlando, 2014. 120–131