Dynamic and Adaptive Calling Context Encoding

Jianjun Li
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
lijianjun@ict.ac.cn

Zhenjiang Wang
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
wangzhenjiang@ict.ac.cn

Chenggang Wu[*]
State Key Laboratory of Computer Architecture,
Institute of Computing Technology,
Chinese Academy of Sciences
wucg@ict.ac.cn

Wei-Chung Hsu
Department of Computer Science,
National Taiwan University
Taipei, Taiwan
hsuwc@csie.ntu.edu.tw

Di Xu[†]
IBM Research - China
Beijing, China
xudi@cn.ibm.com

ABSTRACT

Calling context has been widely used in many software development processes such as testing, event logging, and program analysis. It plays an even more important role in data race detection and performance bottleneck analysis for multi-threaded programs. This paper presents DACCE (Dynamic and Adaptive Calling Context Encoding), an efficient runtime encoding/decoding mechanism for single-threaded and multi-threaded programs that captures dynamic calling contexts. It can dynamically encode all call paths invoked at runtime, and adjust the encodings according to program's execution behavior. In contrast to existing context encoding method, DACCE can work on incomplete call graph, and it does not require source code analysis and offline profiling to conduct context encoding. DACCE has significantly expanded the functionality and applicability of calling context with even lower runtime overhead. DACCE is very efficient based on experiments with SPEC CPU2006 and Parsec 2.1 (with about 2% of runtime overhead) and effective for all tested benchmarks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors, Testing tools*

General Terms

Performance, Reliability, Experimentation

Keywords

calling context encoding, adaptive, dynamic analysis

1. INTRODUCTION

Calling context is critical for understanding dynamic behaviors of large programs. It plays an important role in a wide range of software development processes and large applications such as testing [11], debugging and error reporting [12, 16], performance analysis [19], program analysis [20], security enforcement [10], and event logging [5]. For example, programmers frequently examine calling context during debugging in the form of stack backtracing. Stack backtracing is a simple but expensive method, so it is used only when programmers require interactions in debugging or when a program failed with core dumps. Recently, data race detection or debugging for multi-threaded programs consider more than the immediate circumstances of failures [5] and more efforts have been focusing on identifying the root causes of bugs. Analyzing or debugging failures often involves recording runtime information continuously during an execution. For example, memory access information is recorded by dynamic race detectors. Calling context information is critical and helpful in analyzing the detected data races and locating the instructions which contributed to the data race. However, some online methods for computing calling contexts, such as stack-walking and maintaining the current location in a calling context tree, are too expensive in terms of time and space to be commonly adopted. As a result, most tools resorted to record only static program locations. Calling context information is also proved to be very helpful for event log analysis. In [21], calling context information reduces the events logged, and after removing redundant events in the replay log, the replay could be much faster. In this improved event logging process, context information was collected through stack-walking. This would incur significant runtime overhead. Therefore, a more efficient calling context identification method for multi-threaded applications is called for.

To identify the dynamic execution path efficiently, methods based on path encoding algorithm [3, 18, 1] are proposed. However, these methods use static encoding so that they work only on complete call graph. For applications

[*]To whom correspondence should be addressed.

[†]This work was done when Di Xu attended Institute of Computing Technology, CAS.

CGO '14 February 15 - 19 2014, Orlando, FL, USA

ACM 978-1-4503-2670-4/14/02 $15.00.

containing indirect branches (indirect jumps and indirect calls), existing methods need to identify the targets of indirect branches using static pointer analysis or training runs. This causes negative impact on practicability, and prevent existing methods from being widely used.

This paper presents DACCE (Dynamic and Adaptive Calling Context Encoding), which can work on incomplete call graph. It can dynamically encode all call paths invoked at runtime, including call paths in dynamically loaded libraries. It does not need source code analyses (for example, points-to analysis) or offline training to assist context encoding. In addition, since DACCE is based on dynamic instrumentations, it can only encode paths that are actually invoked during execution. It also can collect information at runtime, and in return, could be used to support more efficient encoding/decoding for complex applications with numerous calling paths. Compared to existing context encoding methods, DACCE does not require training based static profiling. Since DACCE is implemented as a shared library, it could work with other tools more easily. Some researchers may be concerned about the runtime overhead of dynamic binary instrumentation that DACCE might incur. Our prototype has shown that DACCE yields even lower runtime overhead than existing static encoding methods due to its adaptive encoding approach.

Nowadays, increased parallelism has been the driving force in computing rather than increased clock rates. This ongoing shift toward multicore paradigms has rendered more and more applications multi-threaded. DACCE is designed to serve both multi-threaded and single-threaded programs efficiently.

The main contributions of this work are summarized as follows:

- We propose a dynamic context encoding algorithm which can efficiently encode all function calls invoked at runtime. It can work on incomplete call graph. A decoding algorithm for the dynamic encoding method is also provided.

- We propose an adaptive encoding method which can adjust the encodings according to program's runtime behavior.

- Our work expands the applicability of context encoding methods to more applications.

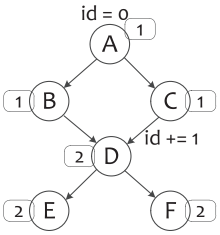
- We have implemented DACCE prototype system and evaluated it using SPEC CPU2006 and Parsec 2.1. Experimental results show that DACCE can be practically applied for both single-threaded and multi-threaded code.

2. MOTIVATION

2.1 Background: Context Encoding

In [3], Ball and Larus proposed an efficient algorithm (BL algorithm) to encode intra-procedural control flow paths taken during execution. PCCE [18] leverages the Ball-Larus (BL) control flow encoding algorithm to encode acyclic contexts.

One simple example of context encoding method is illustrated in Figure 1. In the figure, instrumentation is marked on control flow edges and node annotations (i.e. numbers in boxes) represent the number of calling contexts. In the example, only the edge CD needs to be instrumented by "$id = id + 1$", then the contexts of any point in execution can be clearly distinguished.



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

Figure 1: Example of context encoding

In the existing method, it first computes the number of contexts for each node in a topological order. It then traverses each node n to encode the call edges and insert instrumented code. Each node n is encoded by the numbers in the range of $[0,\ numCC(n)]$, in which $numCC(n)$ represents the number of calling contexts of node n. For each edge $e =< p, n, l >$ (n is the current node, p is the parent of n, and l is the callsite), the instrumentation is added before and after the call site l. Before l, the context identifier id is incremented by the sum of all preceding caller's numCCs; after l, id is decremented by the same amount.

2.2 Our Goal and Issues

In this paper, we aimed to design a context encoding method, which can be easily applied to a larger scope of applications. To achieve this goal, the following issues should be efficiently solved.

Issue 1: *Extra profiling runs and static program analysis caused by indirect calls.*

For static encoding, we need to know the complete call graph before encoding. To identify all targets of indirect calls, points-to analysis is needed in previous approach such as PCCE [18]. Due to the conservative nature of points-to analysis, many possible but unlikely targets would be identified. In real runs, the actual number of targets may be much smaller than what was identified. That means there are many false positives among the identified targets. Large programs with many function pointers have more identified targets, which results in a greater maximum encoding range. For programs written with Object-Oriented Languages, in which indirect invocations are more common, this problem would become more serious. In existing context encoding methods [18] to reduce the runtime overhead incurred by indirect calls, profiling runs are used to identify the likely targets. This adds extra burden on users or application developers.

In many real programs, such as the Apache web server and Firefox, some libraries (software plugins) are dynamically loaded. The functions in these libraries are usually invoked via function pointers. The targets of these function pointers are determined at runtime, so they cannot be identified even with points-to analysis. Because the mapping address of libraries may vary across runs, profiling runs may not work well either. Therefore, we should have a more robust method that can encode the call paths that contain indirect calls dynamically instead of relying on pure static analysis with profiling.

Issue 2: Dynamic loading and call paths within shared libraries.

For function calls to shared libraries, we do not know the real targets of function calls to shared libraries before link time. Because the mapping address of a library varies across runs, we cannot use profiling runs or source code analysis to get the real targets either. Therefore, a dynamic encoding mechanism is needed to encode the function calls via PLT (Procedure Lookup Table). Though the existing method can be extended to encode PLT calls by annotating and encoding the call graph post link time, it will add a large number of library functions in the call graph. Because some library functions (for example, fprintf) are likely to be called from many different call sites, the required encoding space will increase dramatically and result in insufficient encoding space that may exceed the 32 bit integer range. Therefore, a more efficient encoding method is needed to handle the function calls to shared libraries.

Issue 3: Efficient in encoding space and time.

At runtime, only a fraction of call paths will be actually executed. However, static encoding methods will encode all paths in the call graph/control flow graph, this may incur extra overhead and cause insufficient encoding space. To avoid this problem, we need an encoding method which can only encode the call paths actually invoked during execution.

The calling context recording method is usually used in debugging, event logging or analyzing tools. In data race detectors or event logging tools, the correct run must be reasonably fast. Therefore, the overhead of obtaining the runtime calling context should be kept low. To reduce the runtime overhead, existing methods usually use profiling runs to identify the hot and cold call edges. For programs with different execution behavior with varying input datasets or different interferences among threads, static profiling of hot call paths may be inaccurate. These tools may record a large number of context-sensitive events online since they do not know which ones might be relevant for error reporting. To reduce the log file growth, the context sensitivity information must be compressed. Therefore, we would like to build an adaptive method, which requires less work from users and can adjust the encodings according to program's execution behavior.

Issue 4: Avoid interfering with the compilation/optimization of client programs.

To compute the encodings of calling paths at runtime, we need to instrument the client program. Source instrumentation may interfere with compiler optimizations such as inlining and tail call elimination. This will cause the execution behaviors of the instrumented program to be different from real runs. Moreover, it is difficult to obtain the source code of commercial application software. For dynamically linked programs, it is also very difficult to encode the call edges in libraries using source instrumentation. Therefore, we should consider taking other approaches to compute the calling contexts such as dynamic binary instrumentation.

Issue 5: Multi-threaded programs.

For multi-threaded programs, there are several new encoding issues. As discussed in [18], since all functions may operate on the context identifier id, it is declared as a global variable. However, in multi-threaded programs, each thread should have independent calling contexts at runtime. If the context identifier is still globally shared, all threads will operate on the same id. In this case, the context encodings of all threads are added to the same context identifier, and it generates a meaningless or misleading encoded path value. Therefore, each thread should operate on private context identifiers at runtime. A straightforward method is to create a global variable (used as context identifier) for each thread. In this way, the encoding process of all threads would not affect each other. However, to do this, we need to generate separate instrumentation code for each thread. This could lead to code bloat, synchronization errors and other issues.

2.3 Key Challenges

In this paper, we aimed to dynamic and adaptive calling context encoding algorithm. It can dynamically encode all function calls invoked at runtime and adjust the encodings according to program's execution behavior. Its overhead is low and its captured calling context is precise. Such an algorithm has two main challenges:

How to handle newly identified call edges?.

Existing encoding algorithms only work on complete call graph. To encode newly identified call edges at runtime, a new encoding algorithm is needed.

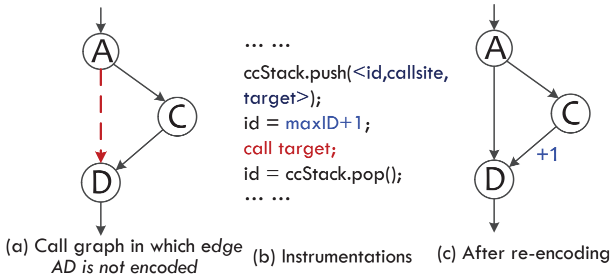
How to ensure the collect path ids be correctly decoded?.

As discussed above, to reduce the time and space overhead incurred by context profiling, we need to adjust the encodings according to application's runtime behavior. After adaptive encoding, the encodings of call edges may change, so the decoding algorithm should ensure that the collected path ids can be correctly decoded.

3. DYNAMIC ENCODING METHOD

DACCE encodes the function call edges invoked at runtime. It starts with a call graph containing only function ”main”. Initially, only the entry function ”main” is instrumented and all function calls (including indirect calls, tail calls, PLT calls, and normal calls) are replaced with instrumentations to invoke a runtime handler. When a function call is called for the first time, the runtime handler will be invoked to take care of a) adding the current call edge to the call graph, and b) inserting code to save and restore the context encodings before and after the instruction call. Function calls inside the target function of the current call are also replaced with ”call to $RuntimeHandler$”. Call edges are added to the call graph after their first invocation, so the call graph expands as the program runs, and only the call edges that are invoked in the current execution are encoded.

The newly invoked call edges are not encoded immediately. Only when the newly added edges exceed a threshold or the frequently executed call paths are not encoded, the whole call graph is re-encoded. The reason not to encode newly called edges is to minimize encoding time, which is part of the runtime. We perform re-encoding only when sufficient information is gathered. This follows the principle



```
… …
ccStack.push(<id,callsite,
target>);
id = maxID+1;
call target;
id = ccStack.pop();
… …
```

(a) Call graph in which edge *AD is not encoded* (b) Instrumentations (c) After re-encoding

Figure 2: Encoding Normal Calls

of dynamic optimization. The complete call path may be divided into sub-paths by the unencoded call edges at runtime. Similar to the way PCCE handles recursive and indirect calls, we use a stack to save the encoding context before invoking unencoded call edges (in this paper, we call the stack *ccStack*). For call paths containing unencoded call edges, the context encoding is composed of current context *id* and the content on *ccStack*.

Since the context encodings of sub-paths are saved in *ccStack*, we must find a way to indicate whether the context encoding on the top of *ccStack* should be popped or not at each round of decoding process. For example, in Figure 2(a), edge AD is not encoded. Assume the current function is D, the current *id*=0, and the *ccStack* has an entry $< 0, A, D >$[1]. This context encodings can be decoded into AD or ACD. While decoding, we need a method to decide which edge (AD or CD) was taken. In [18], to correctly decode the call paths contain recursive or indirect calls, it uses dummy edges to indicate whether there are context encodings of sub-paths saved on *ccStack*. However, it needs a complete call graph before encoding, which is not available with our dynamic encoding approach.

We can see that there is a maximum encoding value (we call it *max*ID for short) after the call graph is encoded. Though the call graph is incomplete in our dynamic encoding approach, the *max*ID is always the maximum possible encoding value at runtime. That is, the possible encoding value at runtime is within the range [0, *max*ID]. Therefore, we can use the numbers in the range of [*max*ID+1, 2**max*ID+1] to indicate if there is an unencoded call edge in current sub-path.

As discussed above, the unencoded call edges would divide the complete path into several sub-paths. The head function of a sub-path must be the target of an unencoded call. In such case, we can set the context *id* to *max*ID+1 and save encoding context (including current callsite and context *id*) before invoking an unencoded call edge. If the sub-path is acyclic, each function would appear at most once in it. Therefore, the target function of the saved callsite is the head function of current sub-path. In this way, the unencoded call edges can be correctly identified. More details are introduced in the following sections. In summary, this novel encoding approach is to effectively deal with incomplete call graphs that the dynamic encoding approach must handle.

3.1 Normal Function Calls

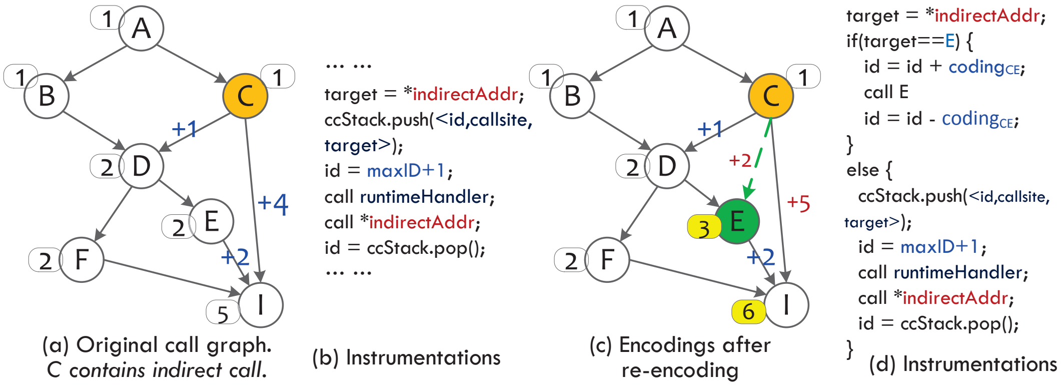
Algorithm 1 Decode a calling context

```
Input:
    id: the encoding id
    ifun: the function at which the encoding was emitted
    ccStack: the content on ccStack when the encoding was emitted
Function AdjustID()
if $id > maxID$ then
    id ← id - (maxID+1)
    onstack ← $true$
end if
Function Decode(id, ifun, ccStack)
onStack ← $false$
AdjustID()
cc ← <ifun,->
while $true$ do
    while id = 0 and onstack=true do
        <id’,cs’,target’> ← ccStack.top()
        if ifun=target’ then
            onstack ← $false$
            e=<p,n,cs> ← getEdge(cs’,ifun)
            if $e$ is a back edge then
                <id, cs, target, count> ← ccStack.pop()
            else
                <id, cs, target> ← ccStack.pop()
            end if
            ifun ← p
            cc ← <ifun, cs, target, (count)> • cc
            AdjustID()
        else
            break
        end if
    end while
    for each e=<p,ifun,cs> ∈ E do
        if En(e) ≤ id < En(e)+numCC(p) then
            ifun ← p
            cc ← <ifun, cs> • cc
            id ← id - En(e)
            break
        end if
    end for
    if CcStack.empty() and e=$null$ and id=0 then
        break
    end if
end while
print $cc$
```

For normal calls, the runtime handler is invoked at their first invocation. Initially, all functions calls are patched with invocations to the runtime handler. The invoked edge is added in the call graph, but that edge is not encoded until the next re-encoding process. After that, instrumentation code are generated and patched. At the end of the runtime handler, the control will return to the newly generated code, and the call instruction will be executed.

As discussed above, sub-paths that contain unencoded call edge should be encoded by the numbers in the range of [*max*ID+1, 2**max*ID+1]. While decoding, if encoding id is in the range [*max*ID+1, 2**max*ID+1], it indicates that current sub-path is separated by an unencoded call edge. As depicted in Figure 2(b), after instrumentation, the current context id, the callsite and the target (i.e. $< id, callsite, target >$ in the figure) are pushed onto *ccStack* before function call, and *id* is set to *max*ID+1. After the target function returns, *id* is restored to its previous value. In the example shown in Figure 2(a), since the context *id* is set to *max*ID+1 before the function call, the call path after

[1] The *callsite* in the tuple should be the address of the CALL instruction. For simplicity, we use the name of caller function to indicate the callsite in the examples.



(a) Original call graph. C *contains indirect call.*

```
... ...
target = *indirectAddr;
ccStack.push(<id,callsite,
target>);
id = maxID+1;
call runtimeHandler;
call *indirectAddr;
id = ccStack.pop();
... ...
```

(b) Instrumentations

(c) Encodings after re-encoding

```
target = *indirectAddr;
if(target==E) {
  id = id + codingCE;
  call E
  id = id - codingCE;
}
else {
  ccStack.push(<id,callsite,
target>);
  id = maxID+1;
  call runtimeHandler;
  call *indirectAddr;
  id = ccStack.pop();
}
```

(d) Instrumentations

Figure 3: Encoding Indirect Calls

function D would be encoded to a number within [maxID+1, 2*maxID+1]. Besides, the encoding context before invoking function D is also saved.

Consider the example context AD. When A invokes D, the id, $callsite$ and index of function D (the entry address or a unique index of a function) are pushed onto $ccStack$. After id value is pushed, it is set to 1 (in the example call graph, maxID=0, so maxID+1=1). As a result, it is encoded as a sub-path A with id=0 on the $ccStack$ and the encoding of current sub-path D is 1.

If there is more than one unencoded call in the call path, the encodings before all unencoded calls will be saved on $ccStack$. Therefore, any level of unencoded calls can be handled. For example, suppose the full call path is A- ->B->C- ->D, in which edge AB and CD are unencoded call edges. Because AB and CD are un-encoded, the full path ABCD is now divided into sub-path A, path BC and sub-path D. When the edge AB is invoked, the encodings of sub-path A will be pushed on $ccStack$, and id is set to maxID+1. While edge CD is invoked, the encodings of sub-path BC would also be pushed on $ccStack$. Therefore, the full path is encoded with $id = max$ID+1, and with tuples <0,A,B>, $<max$ID+1,C,D> on $ccStack$.

The decoding algorithm is presented in Algorithm 1. It takes the context id, function index, and content of $ccStack$ as part of its inputs. In the algorithm, $ccStack.top()$ is the top entry of $ccStack$. On $ccStack$, each entry has three or four elements, which is the context id, the $callsite$, the index of target function ($target$), and the number of repetitive recursions ($count$). At line 13, function $getEdge(cs', ifun)$ returns the edge which is at callsite cs' and ends with $ifun$. If such edge does not exist, it returns NULL. For edge $e =< p, n, cs >$, p is the caller, n is the target function and cs indicates the callsite.

Intuitively, it decodes one acyclic sub-path at a time, until all encodings saved on $ccStack$ are decoded. In the algorithm, lines 9-25 handle unencoded call edges. An acyclic sub-path is decoded by lines 26-33. We use a flag ($onstack$) to indicate if there is an unencoded call edge in a sub-path. In the decoding process, if the current sub-path encoding is greater than maxID, we will adjust the id and set the $onstack$ flag (as shown in lines 1-4). Since the unencoded call edges divide the complete path into acyclic sub-paths, each function would appear at most once in a sub-path and the head function of a sub-path must be the target of the saved callsite. Therefore, we first try to match the decoded context with the call edge on the top of $ccStack$ in each round of decoding process (as shown in lines 9-11).

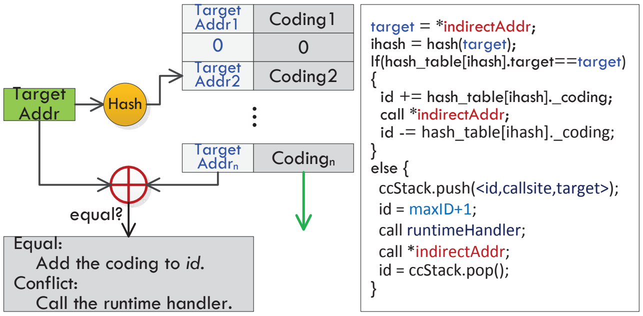
Consider an example of decoding a context. For the call graph in Figure 2, assume the current function is D, the current $id = 1$, and $ccStack$ has an entry $< 0, A, D >$. Since id is greater than maxID=0, the value of id is adjusted and the $onstack$ flag is set. Then, the entry in $ccStack$ is popped since $ifun = target'$, and function A becomes the starting point. After this, the $ccStack$ is empty and $id = 0$, so the decoding process terminates, and calling context AD is decoded.

3.2 Indirect Calls

As discussed above, DACCE does not need points-to analysis or profiling runs to identify the targets of indirect calls since the targets of indirect calls will be identified at runtime. DACCE handles indirect calls in a similar way to normal calls. Initially, the indirect calls are also replaced with instructions that invoke the runtime handler. After their first invocation, the indirect calls are also patched. The instrumented code after their first invocation is shown in Figure 3(b). As shown in the figure, before the indirect invocation, the current context id, the callsite and the target (i.e. $< id, callsite, target >$ in the figure) are pushed onto ccStack, and id is set to maxID+1. After the target function returns, id is restored to its previous value. To identify the targets of indirect calls, the runtime handler is invoked before each indirect invocation. In the runtime handler, the target function node and the corresponding call edge are added to the call graph.

Consider the example context $ACEI$. When C invokes E by an indirect call, the id, $callsite$ and index of function E are pushed onto $ccStack$. After id value is pushed, it is set to 5 (in the example call graph, maxID=4, so maxID+1=5). As a result, taking the remaining path EI leads to id=7. Therefore, the final encoding result is id=7, and the $ccStack$ has an entry $< 0, C, E >$.

This context encoding can be decoded by Algorithm 1. Since the id is greater than maxID, the id will be adjusted and the $onstack$ flag will be set. After the first iteration of the outer loop, the sub-path EI is decoded. In the next iteration, since $id = 0, onstack = true$ and $ifun = target'$, the entry in $ccStack$ is popped and function C becomes the starting point. After this round, the value 0 on $ccStack$ is decoded to the sub-path AC. The two sub-paths constitute the full calling context $ACEI$.



Equal:
  Add the coding to *id*.
Conflict:
  Call the runtime handler.

```
target = *indirectAddr;
ihash = hash(target);
If(hash_table[ihash].target==target)
{
  id += hash_table[ihash]._coding;
  call *indirectAddr;
  id -= hash_table[ihash]._coding;
}
else {
  ccStack.push(<id,callsite,target>);
  id = maxID+1;
  call runtimeHandler;
  call *indirectAddr;
  id = ccStack.pop();
}
```

Figure 4: Instrument the indirect calls

After re-encoding, the identified targets of indirect calls are encoded separately (as depicted in Figure 3(d)). This instrumentation method is effective with a small number of indirect targets. If an indirect call has a large number of targets, it will probably incur many extra comparisons for each invocation. We have observed such cases in several benchmarks, such as 400.perlbench, 445.gobmk, and so on. To avoid the overhead incurred by extra comparisons, a new instrumentation method was developed (as shown in Figure 4).While instrumenting an indirect call, if the number of identified targets exceeds a threshold, the identified target addresses and corresponding coding of the indirect call edges will be saved in a hash table. At runtime, the instrumented code will first calculate the hash value by the target address of current invocation. Then the value saved in the hash table is compared with the current target address. If they are equal, the coding saved in hash table are added to the current *id*; if not, the encodings are saved in *ccStack* and runtime handler is invoked.

DACCE identifies invocation targets dynamically, so it avoids many false positive targets affecting the encodings. Compared with PCCE, our method has less instrumentation and requires a smaller encoding space. Once again, this method does not need any program analysis or profiling runs to deal with indirect call targets.

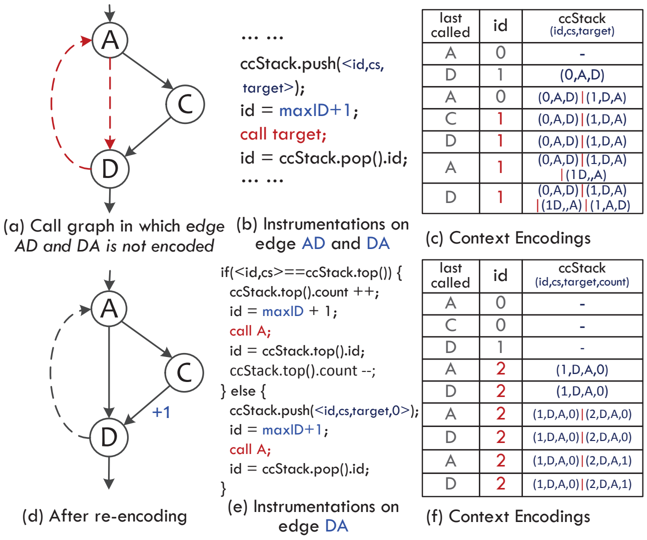
3.3 Recursive Calls

Figure 5(a) gives an example call graph which contains one recursive call. Since we do not know whether the call edge is a recursive call, the recursive call is treated as a normal call at its first invocation.

Consider the context $ADACDAD$. It is encoded as $id = 1$ and 4 entries (<0,A,D>, <1,D,A>, <1,D,A> and <1,A,D>) on *ccStack*. Since the current *id* is greater than maxID, so the *id* will be adjusted and the *onstack* flag will be set. After that, we can get $id = 0$. Next, sub-paths AD and DA will be decoded by lines 9-25 of the algorithm, and sub-path ACD will be decoded by lines 26-33. In the next iteration, sub-paths DA and AD are decoded. Eventually, we obtain the calling context $ADACDAD$.

As discussed in section 3, to correctly decode the full call path, we should ensure that each sub-path is an acyclic path. That is, the full call path should be divided into several sub-paths by unencoded and recursive calls. Therefore, the recursive calls will not be encoded while re-encoding the call graph.

For highly repetitive recursive calls, the *ccStack* could grow very fast. This will incur runtime overhead, as well as space overhead for storing the collected calling contexts. In DACCE, we compress the repetitive paths in ccStack with a counter which records the number of repetitions. As shown in Figure 5(e), while encountering a recursive call at runtime, the value of *id* and *callsite* in the top entry of *ccStack* are compared with current *id* and *callsite*. If they are equal, the code to increment the counter ($ccStack.top().count{+}{+}$) will be executed; otherwise, the code to push the current *id* and *callsite* onto *ccStack* will be executed. In both cases, the id will be set to maxID+1 before the recursive call.



(a) Call graph in which edge *AD and DA is not encoded*

```
… …
ccStack.push(<id,cs,
 target>);
id = maxID+1;
call target;
id = ccStack.pop().id;
… …
```

(b) Instrumentations on edge AD and DA

last called	id	ccStack (id,cs,target)
A	0	-
D	1	(0,A,D)
A	0	(0,A,D)|(1,D,A)
C	1	(0,A,D)|(1,D,A)
D	1	(0,A,D)|(1,D,A)
A	1	(0,A,D)|(1,D,A) |(1D,,A)
D	1	(0,A,D)|(1,D,A) |(1D,,A)|(1,A,D)

(c) Context Encodings

(d) After re-encoding

```
if(<id,cs>==ccStack.top()) {
  ccStack.top().count ++;
  id = maxID + 1;
  call A;
  id = ccStack.top().id;
  ccStack.top().count --;
} else {
  ccStack.push(<id,cs,target,0>);
  id = maxID+1;
  call A;
  id = ccStack.pop().id;
}
```

(e) Instrumentations on edge DA

last called	id	ccStack (id,cs,target,count)
A	0	-
C	0	-
D	1	-
A	2	(1,D,A,0)
D	2	(1,D,A,0)
A	2	(1,D,A,0)|(2,D,A,0)
D	2	(1,D,A,0)|(2,D,A,0)
A	2	(1,D,A,0)|(2,D,A,1)
D	2	(1,D,A,0)|(2,D,A,1)

(f) Context Encodings

Figure 5: Encoding recursive calls

4. ADAPTIVE ENCODING METHOD

To further reduce the runtime overhead of DACCE, we propose an adaptive encoding method, which adjusts the encodings according to collected calling contexts. At runtime, some of the collected contexts will be decoded to extract runtime information. The re-encoding process will be initiated when the following cases are detected.

- The number of identified call edges reaches a threshold.
- The frequently invoked call paths have changed.
- The ccStack is frequently accessed.

For multi-threaded programs, we should first stop the execution of all threads. In our system, we register a signal handler for all threads in the process to suspend the threads at runtime. After the program is suspended, the collected contexts are analyzed and the call graph re-encoded. In the adaptive encoding procedure, the following actions will be taken:

- ◇ Decode the collected contexts, mark the frequently invoked call edges.
- ◇ Encode the whole call graph, and adjust the encodings according to the invocation frequency. The most frequently invoked edge will be encoded with 0. The edges encoded with 0 do not need instrumentation, the runtime overhead will be reduced.

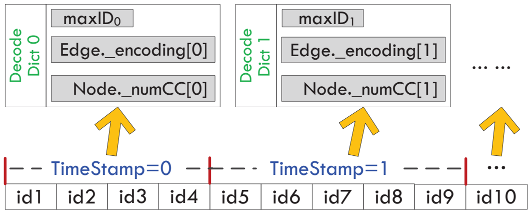


Figure 6: Decoding mechanism with re-encoding

- ◇ Analyze the contents on ccStack of collected contexts. If they are highly repetitive, adjust the encoding algorithm on recursive calls to compress the saved contexts on ccStack.
- ◇ Instrument the program with the new encodings.

After instrumentation, the current id and entries on *ccStack* are regenerated according to the new encodings. The return address of all active functions on the stack should be modified to corresponding addresses in newly generated instrumentation. After the adaptive encoding process, all threads will be resumed, and the program will continue executing.

4.1 Decoding mechanism with re-encoding

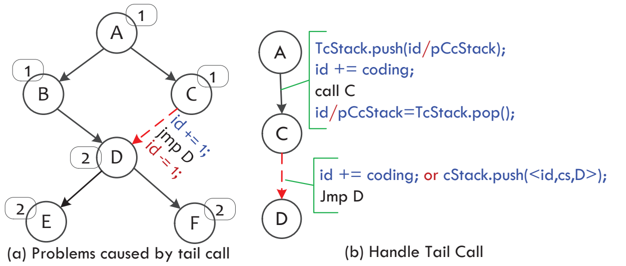
With adaptive encoding method, the call graph is growing dynamically as the program runs. To correctly decode a recorded context, we need the exact call graph and the encoding information when the context is recorded. Therefore, we use a global timestamp (*gTimeStamp*) to tag the collected contexts. After each re-encoding process, *gTimeStamp* will be incremented by 1 to indicate the call graph has been changed. Figure 6 illustrates the decoding mechanism. All collected contexts are tagged with *gTimeStamp*. In our system, the information that is needed when decoding a context *id* is store in *Edge* and *Node* structures respectively. *Edge._encoding* is the encodings of an edge, and *Node._numCC* is the number of contexts of a function. $maxID_n$ is the maximum path encoding value for the call graph when *gTimeStamp* is n. These three elements constitute a decoding dictionary. While decoding, we will use *gTimeStamp* to select the respective decoding dictionary.

5. IMPLEMENTATION ISSUES

5.1 Function calls to shared libraries

For function calls via PLT (we call them PLT calls in this paper), we can encode these call edges at runtime after all needed libraries have been loaded. Since some library functions (for example, fprintf) are likely to be called from many different call sites, the maximum number of contexts could exceed the allowable encoding range.

In our work, the way to handle PLT calls is similar to normal calls. The PLT call edges are added in the call graph only as needed at runtime. Initially, all PLT calls are patched with invocations to the runtime handler. In the handler, we first get the real target address of the PLT call. The edge between the callsite and the real target is added to the call graph, but that edge is not encoded until the next re-encoding process. After that, code to save and restore the encoding context (as shown in Figure 2(b)) are generated and patched.


A
1
B
1
C
1
D
2
E
2
F
2
id += 1;
Jmp D
id -= 1;
A
TcStack.push(id/pCcStack);
id += coding;
call C
id/pCcStack=TcStack.pop();
C
id += coding; or cStack.push(<id,cs,D>);
Jmp D
D

(a) Problems caused by tail call

(b) Handle Tail Call

Figure 7: Handling tail calls

5.2 Tail Calls

In compilers, tail call elimination is a common optimization. To compute the encodings, instructions are inserted before and after function calls. If the encoding algorithm is implemented on source level, tail call optimizations of compiler will not take place. Therefore, in several existing related works, such as PCCE [18], there are no tail calls in the selected test programs.

In Figure 7(a), CD is a tail call. Assume the invocation order is *ACDFABDF*. Because function *D* will return directly to function *A* instead of *C*, the instrumented code ”*id*-=1” will not execute. After *ACDF* is returned, the encoding value in *A* is 1. This will lead to incorrect encodings for subsequent call paths. For example, path *ABDF* would be encoded as 1 while the correct encoding should be 0.

Our solution is presented in Figure 7(b). Analyzing the context encoding method, we can observe that the encoding context is identical before and after an invocation. For tail call edges that cannot be encoded with 0, we store and restore the encoding context in the caller of the function which contains a tail call. We use another helper stack (we call it *TcStack* in this paper) to save encoding values before a tail call. Unlike *ccStack*, the content in *TcStack* is not required for decoding the contexts, so it will not increase the overhead while recording the calling contexts. As illustrated in Figure 7(b) (in the example, edge CD is a tail call), instructions to store and restore the encoding context before tail call are instrumented on edge AC.

Like normal calls, tail calls are added to the call graph after their first invocations. In the runtime handler, we modify the instrumented code of current function's caller and update the *TcStack*. Before re-encoding, the instrumented code of tail calls would be ”ccStack.push(<id,cs,target>)”. In this case, the address of *ccStack*'s top entry would be also stored in *TcStack*.

To handle tail calls via indirect branches, all indirect branches are also instrumented to get the branch targets. If the target of an indirect branch is out of current function, that indirect edge will be treated as tail call.

5.3 Encoding Multi-threaded Program

To correctly encode a multi-threaded program, we utilize thread local storage (TLS) to compute the context identifier at runtime. TLS allocates variables so that there is one instance of the variable per extant thread. TLS is supported by most architectures, such as IA-32, X86-64, IA-

64, SPARC, Alpha etc. For example, on X86-64, the thread-local variable can be accessed by %gs:tlsoffset (for 32-bit program) or %fs:tlsoffset (for 64-bit program). In this way, we can only instrument one copy of code, and guarantee that each thread operates on its own context id independently.

The *ccStack* used to encode the recursive path and indirect invocation path should also be allocated within the TLS section. However, the required size of *ccStack* may vary widely for different threads. To reduce the space overhead, *ccStack* should be created sparingly. Initially, we don't allocate memory for *ccStack* (the pointer variable which points to the *ccStack* should be within TLS section). Therefore, if *ccStack* is accessed for the first time, an access violation trap will be raised. In the trap handler, we allocate memory space for *ccStack*. Furthermore, the bottom of *ccStack* is protected to detect stack overflow.

To recover the full calling contexts of multi-threaded program, we collect thread creation and work migration information at runtime. For Pthreads programs, we will intercept function *clone* and record the thread creation information at runtime. While decoding the contexts, the sub-path to create the current thread is also decoded.

6. EVALUATION

6.1 Experimental framework

We implemented DACCE as a shared library (*dacce.so*) on Linux. We use the environment variable *LD_PRELOAD* to intercept the *__libc_start_main* routine (within which the main function is called). In the intercepted routine, a call graph that contains only function ”main” is constructed, and the function ”main” is instrumented to capture all function calls invoked in ”main” at runtime. After that, the real *__libc_start_main* routine is invoked and the execution starts.

To verify the correctness and effectiveness of DACCE, we implement a sample module with libpfm4 [9] in our system. Libpfm4 is a helper library for the performance monitoring events provided by the hardware or the OS kernel. In our system, we periodically sample the program and record the context identifiers at the sample points. We also capture the calling contexts with a stack-walking method. The calling contexts obtained by the two methods are cross validated after program ends.

To compare with PCCE [18], we also simulate PCCE in our system. We first use Pin [13] to profile the targets of indirect calls and the invocation frequency of all edges with the same input as in real runs to give PCCE a full potential of profiling. In real runs, we will first add indirect edges in the call graph and adjust the encoding of call edges according to the profiled data.

6.2 Platform and benchmarks

All experiments are performed on a 2-way Intel Xeon server, and each processor is a 1.87GHz Intel Xeon E7-4807. The SPEC CPU2006 and Parsec 2.1 [4] are used as the benchmarks to evaluate DACCE. The SPEC CPU2006 benchmarks were compiled by the Intel C++ Compiler 11.0, and the ”-ipo -O3 -no-prec-div -xSSE4.2” compiler optimization option is used. For Parsec 2.1, the gcc-pthreads version of the precompiled binary distributions is used. In the experiments, SPEC CPU2006 benchmark suite uses the *ref* input set, and Parsec 2.1 benchmark suite uses the native input set.

6.3 Characteristics of benchmarks

Table 1 presents the characteristics of the benchmarks used to evaluate DACCE. For each program, the table lists the number of nodes (Nodes) and edges (Edges) in the call graph, the maximum ID (i.e. context identifier) required in the call graph (MaxID), the frequency of *ccStack* operations (ccStack/s), and the average depth of *ccStack* (depth) for both PCCE and DACCE. Column ”gTS” (abbreviation of gTimeStamp) and ”costs” list the number of times that the re-encoding process is triggered and the time cost in re-encoding processes respectively. The last column ”calls/s” shows the invocation frequency of function calls at runtime.

We can observe that the *ccStack* are frequently accessed (due to recursive calls and indirect calls) in some programs, such as 400.perlbench, 483.xalancbmk, and 453.povray. In our system, we use a 64bit context identifier. For PCCE, there are still several benchmarks which could not be encoded within a 64-bit ID like 400.perlbench and 403.gcc. To fit the *max*ID of these programs into 64-bits, some edges that are never invoked in real runs (according to the profiled data) are deleted. As the data shows, the required encoding space is significantly reduced with DACCE since only the contexts that are invoked at runtime are encoded.

For most benchmarks, the average depth of *ccStack* is 0. This means that the call path can be encoded into one ID without using the *ccStack* most of the time. However, 445.gobmk and 483.Xalancbmk have several recursive invocations occurring on the frequently executed path, so the ccStack has a nontrivial depth.

6.4 Experimental Results of DACCE

The runtime overhead of PCCE and DACCE is shown in Figure 8. As the data shows, for most benchmarks, the runtime overhead of PCCE and DACCE are comparable. However, for 400.perlbench, 483.xalancbmk and x264, PCCE incurs higher overhead than DACCE even when profiled with the same input sets as in real runs. We can observe from Table 1 that the *ccStack* is more frequently accessed with PCCE for 400.perlbench and 483.xalancbmk. This is because edges that are never invoked in real runs may still cause some edges to be identified as back edges in a complete call graph. In DACCE, only the edges invoked at runtime are added in call graph, so the cold edges will not affect the encodings of hot edges. For x264, the reason is that several frequently invoked indirect calls have a large number of targets. In section 3.2, we have discussed that if an indirect call has a large number of targets, it may incur many extra comparisons for each invocation. In DACCE, an instrumentation method was developed to cope with this issue (as introduced in section 3.2).

For some benchmarks, such as 458.sjeng, 433.milc and 434.zeusmp, the overhead of DACCE is slightly higher than PCCE. As illustrated in section 6.1, PCCE depends on static profiling to drive encoding. For these benchmarks, the profiles used are very representative. DACCE uses dynamic profiling which incurs some runtime overhead. Therefore, for benchmarks whose offline profiling is representative, the advantage of using dynamic profiling diminishes. However, even in such cases, the runtime overhead incurred in of DACCE is still relatively low.

As shown in the figures, the average runtime overhead

Table 1: Characteristics of SPEC CPU2006 and Parsec 2.1

	PCCE					DACCE							calls/s
	Nodes	Edges	MaxID	ccStack/s	depth	Nodes	Edges	MaxID	ccStack/s	depth	gTS	costs(us)	
400.perlbench	1468	21065	overflow	4969345	0.20	684	3911	1.4E+11	3095100	0.20	23	1747514	29205101
401.bzip2	122	321	833	0	0.00	50	109	61	38753	0.05	5	3475	7687097
403.gcc	3944	50690	overflow	0	2.94	1931	11518	7.0E+13	315406	0.00	110	2866850	14710894
429.mcf	69	126	53	0	0.00	11	12	3	2069	0.01	2	166	295581
445.gobmk	2273	13687	3.4E+15	246782	2.42	1378	4808	2.4E+11	250321	2.47	76	1732161	13355556
456.hmmer	249	1618	56401	3082	0.00	70	174	42	481	0.02	2	1420	1872530
458.sjeng	139	678	33088	0	0.00	54	232	2945	233	0.00	23	19560	18248384
462.libquantum	118	846	1202640	0	0.00	29	49	15	1	0.01	9	722	44
464.h264ref	398	2698	1.8E+07	424979	0.00	201	1048	34293	5310	0.00	10	84556	7080183
471.omnetpp	1706	11981	1.2E+07	302097	0.11	506	4135	8654	149146	0.04	11	205585	11656043
473.astar	139	469	3177	0	0.00	60	140	101	10606	0.03	10	1922	129559
483.xalancbmk	12535	40392	3.8E+14	4375862	6.91	2170	7321	1422838	596197	6.01	27	3551342	25341805
410.bwaves	369	2189	7248401	0	0.00	82	164	73	2639	0.01	6	433	263845
416.gamess	2442	50080	1.1E+15	0	0.00	362	2017	112645	21925	0.03	19	41810	3390329
433.milc	177	667	5761	0	0.00	57	185	455	46156	0.09	38	524072	380448
434.zeusmp	416	3598	2.9E+08	0	0.00	118	528	5026	485	0.05	81	9640	1601
435.gromacs	619	2919	351721	0	0.00	154	402	1553	5132	0.01	8	4742	919287
436.cactus	876	6394	8552489	0	0.00	271	1533	119729	3003	0.01	3	16197	4662
437.leslie3d	434	3247	6.0E+07	0	0.00	106	597	388	475	0.00	2	880	85206
444.namd	176	482	361	0	0.00	61	101	31	19426	0.02	20	4260	737925
447.dealII	9935	30204	254161	280	0.12	792	3369	1132	16331	0.06	47	30871	19533456
450.soplex	784	1954	96457	2590	0.00	225	453	367	32681	0.07	7	8706	312430
453.povray	1644	12056	8.7E+16	270387	0.84	548	2201	548645	69109	0.76	6	113456	34335309
454.calculix	1009	8307	1.0E+09	0	0.00	416	1660	3043	62812	0.06	11	13485	3662033
459.GemsFD	517	5076	5.1E+08	0	0.00	175	2067	10756	32749	0.01	7	7690	1579372
465.tonto	2144	34717	4.3E+14	0	0.33	657	4548	134983	26186	0.03	101	154889	9545304
470.lbm	75	135	53	0	0.00	13	16	4	0	0.00	3	222	2964
481.wrf	1367	17330	4.5E+12	0	0.00	660	5483	713767	20138	0.03	4	63147	2358117
482.sphinx3	273	1570	27121	0	0.00	134	404	92	4187	0.00	6	1825	1875791
blackscholes	12	26	4	0	0.00	3	5	5	68	0.00	11	644	14646244
bodytrack	1310	11047	151775	0	0.00	218	894	667	68268	0.01	5	12204	6928160
facesim	6213	24377	1.8E+10	0	0.00	264	1102	1104	24132	0.00	5	11029	8891290
ferret	1987	25270	7.9E+14	0	0.00	354	1612	3398	44682	0.00	4	8972	4439120
raytrace	7911	24577	6.8E+08	0	0.02	177	632	235	370	0.06	5	5631	3516574
swaptions	2173	6372	2.6E+08	0	0.00	15	136	51	3	0.03	12	45821	21753118
fluidanimate	2168	6420	2.8E+08	0	0.00	73	144	31	49	0.00	8	23648	76287
vips	5395	25302	7.7E+11	0	0.00	482	1555	26117	3865	0.00	5	3271	855060
x264	820	3299	1079001	0	0.00	221	1052	2017	15729	0.00	4	84911	23984355
canneal	2191	6733	3.4E+08	0	0.00	107	225	44	380	0.00	6	105133	2276649
dedup	121	256	65	0	0.00	21	30	5	30239	0.00	4	7201	1305985
streamcluster	2182	6336	2.6E+08	0	0.00	11	29	15	14	0.00	6	156324	111153

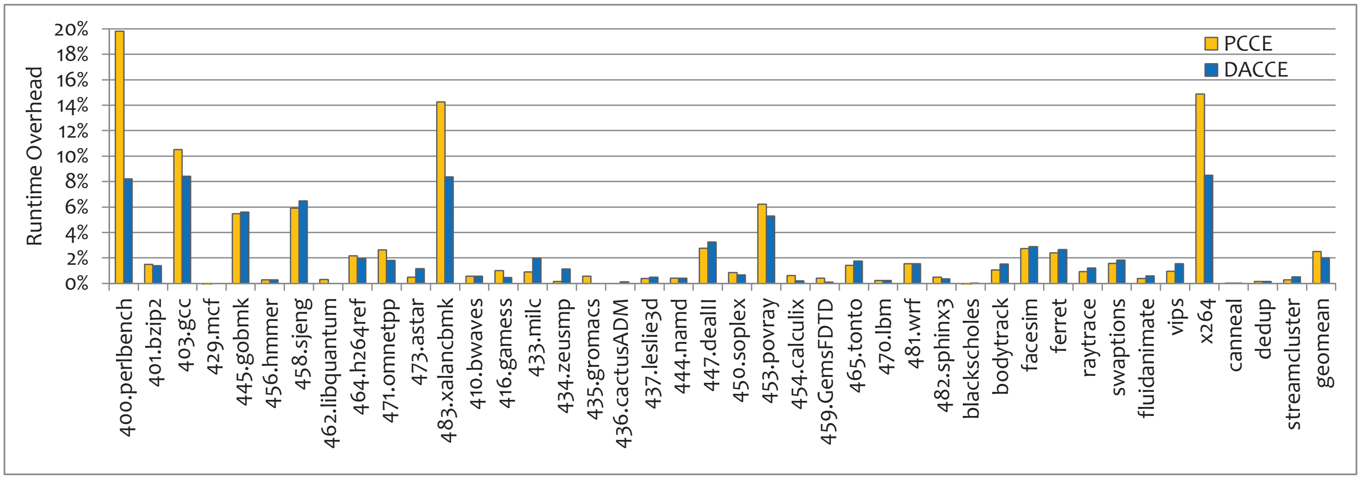


Figure 8: The runtime overhead for SPEC CPU2006 benchmarks and Parsec2.1

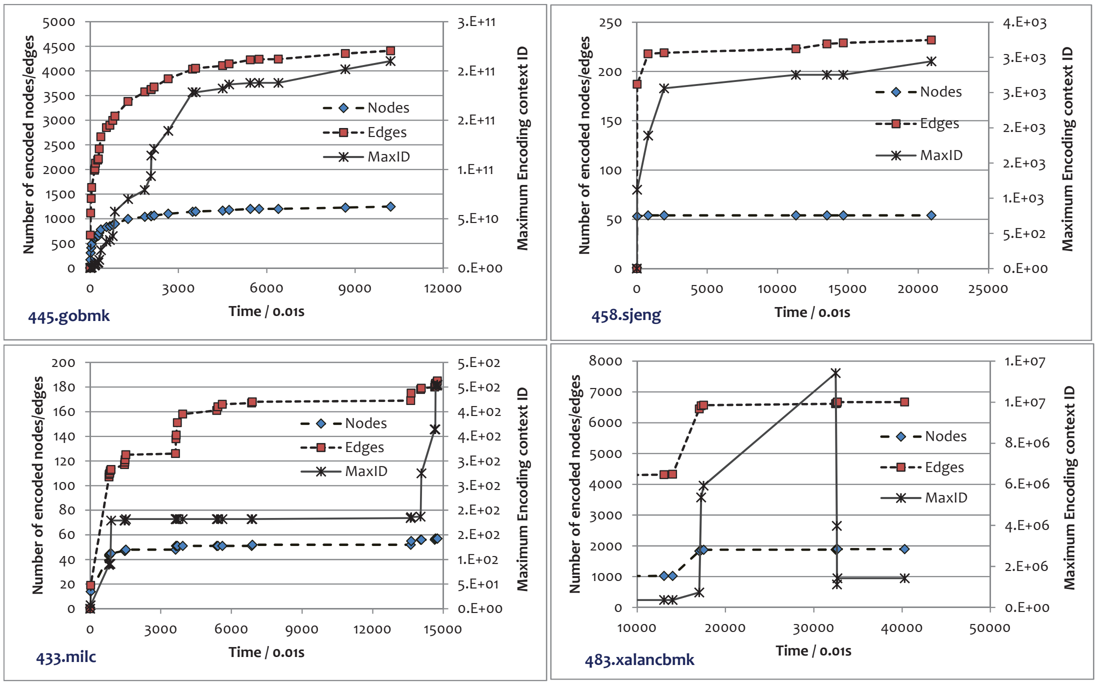


Figure 9: The progress of encodings with DACCE method

is about 2.5% and 2% for PCCE and DACCE respectively. There are several benchmarks which have a relatively higher overhead. Compared to Table 1, we can see that the programs which suffer significant performance degradation are exactly the programs which have high frequency of function call invocations and *ccStack* operations. That is, the overhead of PCCE and DACCE are closely related to the invocation frequency of function calls, especially the invocation frequency of recursive calls.

Figure 9 shows the progress of encodings using DACCE. In the figure, the x-axis is the number of collected context encodings. Since we collect the context encodings periodically, it can also represent the execution time of the program (the sampling period is about 0.01s). The y-axis is the number of encoded nodes/edges and the maximum encoding context ID respectively. Due to space limitation, we only present the data for four representative benchmarks in Figure 9. As the data shows, the re-encoding process is triggered slightly more frequently at the beginning, and the encoding reaches a relatively steady-state quickly. As the program executes, the encodings will be adjusted when hot call paths are changed or new call paths are invoked.

It is very interesting that the value of maximum encoding context ID decreases after the 12th re-encoding process for 483.xalancbmk. After analyzing the log file, we found that one newly identified call edge forms a recursive path in the call graph, and it causes another edge to be identified as a back edge. Coincidentally, that back edge is contained in the call path with the maximum encoding context ID. Since that back edge is not encoded in the later re-recoding process, the value of maximum encoding context ID decreases.

Figure 10 shows the cumulative distributions of the depth of the call stack (i.e., the length of full call path) and ccStack. We present several representative results. In the figure, each diagram corresponds to one selected benchmark. The x-axis is the stack depth needed to represent the calling context and y-axis shows the cumulative percentage of dynamic context instances during execution of which the depth is smaller than the given stack depth. If a curve ascends to 100% very quickly, it means that most contexts for that benchmark could be stored within a small space. The graph for 459.GemsFDTD is typical for most programs. The *ccStack* is always empty while the depth of the call stack gradually ascends. For 483.xalancbmk, the stack depth needed to cover 90% of contexts is about 7200, which is large. This is because the recursive call paths are invoked frequently in 483.xalancbmk, and it results in a relatively larger call stack depth.

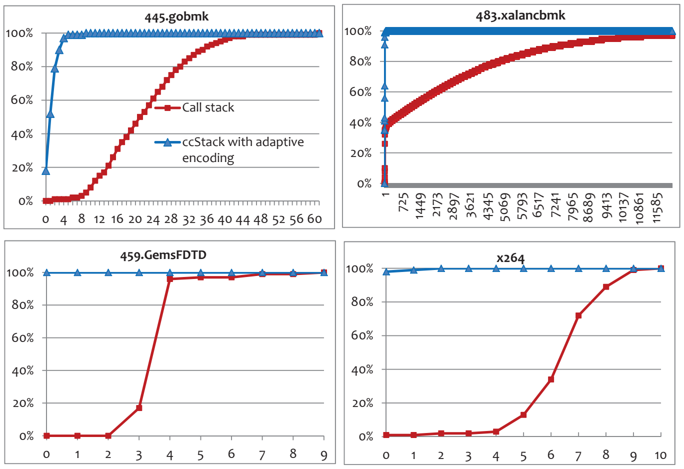


Figure 10: Cumulative distributions of the stack depth

7. RELATED WORK

A straightforward method for capturing the current calling context is stack-walking. For example, Valgrind uses stack-walking method at each memory access to record its context-sensitive program location, and reports this information in the event of a bug [15, 16]. In HpcToolKit [19], the stack-walking method is also used to obtain the calling context at sample point. However, walking the stack frequently would incur significant runtime overhead. An alternative to stack-walking is to build a calling context tree (CCT) dynamically [2, 17, 8]. The program's current position in the CCT is tracked during execution. CCT is very useful in profiling, but it adds a factor of 2 to 4 to program execution time. These overheads are unacceptable for most deployed systems. Recently, sampling-based approaches [22] are used to reduce the overhead. These works keep overhead low by identifying the calling context infrequently, and it is useful for identifying hot calling contexts. However, for applications need coverage of both hot and cold contexts, such as testing and debugging, sampling-based approaches are not appropriate.

In [14], program counter and stack depth are used to identify a calling context. This method has essentially no runtime overhead, but there may be many ambiguous context IDs. To disambiguate the conflict context IDs, the height of stack frames is resized to differentiate conflicting stack heights. It relies on training runs to build an offline dictionary and any new contexts observed online cannot be correctly decoded.

Another approach that trades accuracy for reduced overhead is to compute a probabilistic that only calls context (PCC) [7]. It instruments function calls to compute a hash value that represents the current calling context. The context information obtained by this method is probabilistic and is likely to provide a unique identifier for context. However, without runtime information, it is difficult to map the context identifier to call paths. Recent work [5] presents a method to reconstruct the calling context from hashed numeric identifiers. It uses the static call graph and dynamic information to decode the calling context. To reduce the runtime overhead, it only collects dynamic information at infrequently executed callsites. However, this may cause reconstruction to fail. On average, the runtime overhead is 10% to 20%.

Precise Calling Context Encoding (PCCE) [18] is a recently proposed encoding method to uniquely represent the current context of any execution point using a number of integer identifiers (IDs). It adopts the efficient path profiling encoding approach proposed by Ball and Larus [3] and adjusts it for call path encoding. PCCE offers a non-probabilistic approach. The runtime overhead of PCCE is very low. However, it is a static encoding method. PCCE requires points-to analysis and static profiling to take care of insufficient encoding space and handle function pointers, and it cannot encode the call paths in dynamically loaded libraries. To handle insufficient encoding space and reduce the overhead, profiling runs are needed. This adds extra burden on users, and makes it difficult to use. Moreover, its source level instrumentation approach would interfere with compiler optimizations. Therefore, the application of PCCE in practice is rather limited.

In practical path profiling [6], it uses the encoding space $[N+1, 3N+1]$ to encode cold paths (N is the maximum encoding *id* after deleting cold edge in the control graph). The computed path id of a cold path is not unique. In our paper, we use encoding space [maxID+1, 2*maxID+1] in a different way. In DACCE, a full call path may be divided into several sub-paths, and we use the numbers in the range of [maxID+1, 2*maxID+1] to indicate if there is an unencoded call edge in the current sub-path. The path id for a sub-path is unique, and it can be decoded correctly.

8. SUMMARY AND CONCLUSION

In this paper, we propose DACCE, a dynamic and adaptive encoding method, which can encode the calling contexts of both single-threaded and multi-threaded programs. DACCE is built on top of existing context encoding algorithms but differs from them. Existing algorithms must deal with complete call graphs, while DACCE deals with incomplete call graphs with adaptive encoding. DACCE adopts dynamic binary instrumentation so that the power of calling contexts can be expanded to more applications including applications linked with dynamically shared libraries, multi-threaded and single threaded applications coded in various programming languages. Furthermore, DACCE can avoid limitations of source code instrumentation such as the impact on compiler optimizations. Due to its dynamic nature, DACCE avoids training runs collecting static profiles as static encoding methods do. With expanded functionalities, most readers might be concerned whether a much higher runtime overhead would appear. DACCE has effectively used dynamic profiles of indirect call targets to handle indirect call paths and used compression techniques to handle recursive calls. Its runtime overhead is even less than the PCCE encoding based on the test of our DACCE prototype with SPEC 2006 and Parsec 2.1. We are currently integrating DACCE with several debugging and performance analysis tools as well as some open source tools for multi-threaded applications. We believe with DACCE, the benefit of precise calling contexts can greatly influence many software development and testing tools.

9. ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (NSFC) under grant 61303052. This work was also partly supported by the NSFC under grants 61332009, 61303051, and 60925009, the National High Technology Research and Development Program of China under grant 2012AA010901, the Innovation Research Group of NSFC under grant 61221062, and the National Basic Research Program of China under grant 2011CB302504.

Wei Hsu was partly supported by the National Science Council of Taiwan, ROC, under grants 102-2220-E-002 -031 and 102-2219-E-002 -025.

10. REFERENCES

[1] Profiling all paths: A new profiling technique for both cyclic and acyclic paths. *Journal of Systems and Software*, 85(7):1558 – 1576, 2012.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 85–96, 1997.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, 1996.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, 2008.

[5] M. D. Bond, G. Z. Baker, and S. Z. Guyer. Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 13–24, 2010.

[6] M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 205–216, 2005.

[7] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 97–112, 2007.

[8] D. C. D'Elia, C. Demetrescu, and I. Finocchi. Mining hot calling contexts in small space. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 516–527, 2011.

[9] S. Eranian. Perfmon2: the hardware-based performance monitoring interface for linux. http://perfmon2.sourceforge.net/pfmon_usersguide.html, 2011.

[10] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 62–71, 2003.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, 2005.

[12] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 101–111, 2007.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[14] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred call path profiling. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 175–190, 2009.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007.

[16] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, 2005.

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

[18] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, pages 525–534, 2010.

[19] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 441–452, 2009.

[20] X. Zhang, A. Navabi, and S. Jagannathan. Alchemist: A transparent dependence distance profiling infrastructure. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 47–58, 2009.

[21] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 81–91, 2006.

[22] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 263–271, 2006.