CCCP – Complete Calling Context Profiling in Virtual Execution Environments

Philippe Moret
Faculty of Informatics
University of Lugano
Switzerland
philippe.moret@lu.unisi.ch

Walter Binder
Faculty of Informatics
University of Lugano
Switzerland
walter.binder@unisi.ch

Alex Villazón
Faculty of Informatics
University of Lugano
Switzerland
alex.villazon@lu.unisi.ch

Abstract

Calling context profiling is an important technique for locating hotspots in programs. The prevailing data structure is the Calling Context Tree (CCT) that provides dynamic metrics for each calling context. Existing approaches to calling context profiling in Java either limit portability due to the use of native code or of a modified Java Virtual Machine, create incomplete and inaccurate CCTs, or cause excessive overhead. In this paper, we introduce Complete Calling Context Profiling (CCCP), a new approach that reconciles completeness and accuracy of the created CCTs, portability, and moderate overhead. CCCP relies on a generic bytecode instrumentation framework ensuring comprehensive bytecode coverage, including also the standard Java class library. In order to reduce the overhead of accessing the current CCT node, CCCP transforms code such that the caller passes its CCT node to the callee as a special method argument, while ensuring compatibility with native code, reflection, and stack introspection. We use the resulting CCTs for a detailed analysis of the dynamic behavior of Java systems and present a thorough analysis of the origin of runtime overheads.

Categories and Subject Descriptors B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids; C.4 [*Performance of Systems*]: Measurement techniques; D.2.8 [*Software Engineering*]: Metrics—Performance measures

General Terms Algorithms, Languages, Measurement, Performance

Keywords Calling context profiling, bytecode instrumentation, native code, Java Virtual Machine

1. Introduction

Calling context profiling is a common technique to analyze the dynamic behavior of programs and to locate hotspots. It yields dynamic metrics separately for each calling context, such as the number of method invocations with the same call stack or the CPU time spent in the calling context. The *Calling Context Tree (CCT)* [2] is a well-known data structure to represent calling context profiles at runtime. There is a large body of related work dealing with different techniques to generate CCTs [2, 4, 25, 20, 27, 9], highlighting the importance of calling context profiling.

Unfortunately, most programming languages provide limited access to calling context information and existing techniques for creating complete calling context information cause high overhead, as stressed in [27, 9]. For example, Java exposes calling context information through the `Throwable` API such that a thread can obtain a trace of its call stack using `new Throwable().getStackTrace()`. However, it is not practical to use the `Throwable` API for calling context profiling, which would require allocating a `Throwable` instance and filling the stack trace upon each method invocation, resulting in excessive overhead. In addition, the stack trace provides only class and method name for each stack frame, but not the method signature, making it difficult to distinguish overloaded methods. Moreover, stack traces may be incomplete on certain Java Virtual Machines (JVMs), since the `Throwable` API gives a lot of flexibility to the JVM implementor.

Several techniques have been proposed for computing calling context by maintaining the current location in a CCT [2, 20]. Unfortunately, such techniques can be very expensive with respect to execution time and memory consumption. Other approaches based on sampling and stack-walking have been proposed [4, 25], but they trade performance for accuracy. A hybrid approach improves sampling-based stack-walking by bursty profiling after stack walking [27]. However, that approach does not create complete CCTs and is based on native code using the JVM Profiler Interface (JVMPI) [21], thus limiting its portability. The Probabilistic Calling Context (PCC) [9] significantly reduces overhead by maintaining a probabilistically unique value representing the calling context in a hash table. Unfortunately, this approach excludes methods of the Java class library from the calling context, severely limiting its applicability for profiling. In addition, it requires a modified JVM and does not consider the presence of native code.

In this paper we introduce *Complete Calling Context Profiling (CCCP)*, a new technique for generating complete and accurate CCTs in a fully portable way using bytecode instrumentation. Each *Java method*[1] is transformed so as to maintain the current CCT node in a local variable. The caller passes its CCT node to the callee, and upon method entry, the callee looks up or instantiates its corresponding CCT node. In this way, the current CCT node is directly accessible in instrumented method bodies, which helps reduce the overhead of calling context profiling. CCCP requires

PEPM'09, January 19–20, 2009, Savannah, Georgia, USA.

[1] In this paper, "method" stands for "method or constructor". The term "Java method" denotes any method that has a corresponding bytecode representation (we do not require that the bytecode is generated by a Java compiler). Because constructors cannot be native, they are always Java methods in our terminology.

the introduction of a special method argument, which may hamper compatibility with native code, reflection, and stack introspection. CCCP takes all these issues into account and has been successfully tested with several state-of-the-art JVMs on different platforms.

In order to ensure completeness of the generated CCTs, every executing method needs to be instrumented, including the methods in the Java class library. However, instrumenting the core classes in the Java class library is difficult [16, 23, 5], because modifications of these classes may break bootstrapping and crash the JVM. Hence, CCCP is based on FERRARI (Framework for Exhaustive Rewriting and Reification with Advanced Runtime Instrumentation) [5], a general-purpose bytecode instrumentation framework supporting the instrumentation of the whole Java class library and of all dynamically loaded classes. While FERRARI guarantees complete bytecode coverage, it cannot instrument native code. In order to ensure accuracy of the generated CCTs, CCCP pays particular attention to native code; native method invocations and callbacks from native code into bytecode are correctly represented in the CCTs.

FERRARI offers a flexible interface enabling different user-defined instrumentation modules (UDIs) written in pure Java to control the instrumentation process. We validated CCCP with two different UDIs, one for *platform-independent profiling* and a second UDI for *cross-profiling*, supporting embedded Java processors as targets. Platform-independent profiles [6] convey only platform-independent metrics [14], such as the number of method invocations, the number of executed bytecodes, or the number of allocated instances for each type. Cross-profiles [7] reflect the execution time metric of the embedded target system. The two aforementioned UDIs, which are based on our prior work [6, 7], benefit from the new instrumentation technique presented in this paper, yielding complete and accurate CCTs.

The original, scientific *contributions* of this paper are twofold:

1. We present CCCP, a new instrumentation for calling context profiling that reconciles completeness and accuracy of the generated CCTs, portability and compatibility with standard JVMs, and reduced overhead. We highlight the intricacy of native code and stack introspection, and explain how CCCP correctly handles these issues.
2. We leverage the generated CCTs to analyze the dynamic behavior of typical Java workloads regarding their use of the Java class library. Furthermore, we give a thorough evaluation of the overhead caused by CCCP, exploring the overhead due to the FERRARI framework and considering the impact of load-time instrumentation performed by the framework.

This paper is structured as follows. Section 2 gives a short overview of FERRARI, our generic bytecode instrumentation framework. Section 3 defines our notion of completeness of CCTs. Section 4 introduces CCCP, elaborates on native code issues, and presents generic instrumentation rules that yield complete and accurate CCTs. Section 5 describes two use cases of our instrumentation, platform-independent calling context profiling and cross-profiling for embedded Java processors. Section 6 details our evaluation results. Section 7 discusses related work. Finally, Section 8 concludes this paper.

2. The FERRARI Framework

FERRARI [5] consists of a static instrumentation tool and a runtime instrumentation agent. These tools are controlled by a user-defined instrumentation module (UDI). FERRARI defines an interface that the UDI has to implement and invokes the UDI through this interface. The UDI may change method bodies, add new methods (with minor restrictions), and add fields (with some restrictions). To this end, FERRARI passes the original class bytes to the UDI and receives back the UDI-instrumented class bytes.

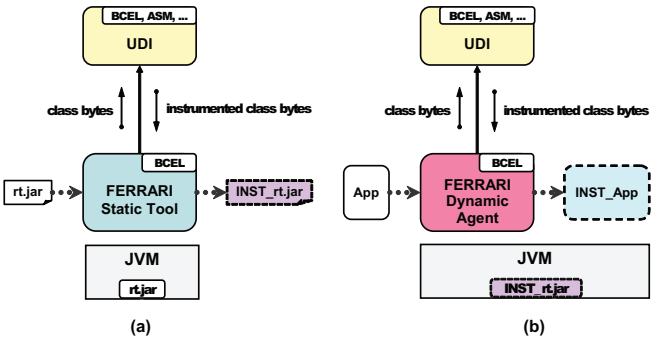

Figure 1. (a) Static and (b) dynamic instrumentation with FERRARI

FERRARI provides a tool to statically instrument the whole Java class library (including vendor-specific classes of the runtime system) according to a given UDI. Figure 1a shows the static instrumentation of the standard Java class library (`rt.jar`), resulting in an instrumented version `INST_rt.jar` that is used by the JVM executing the application under dynamic instrumentation (see Figure 1b).

Application classes are dynamically instrumented at load-time by FERRARI's agent in collaboration with the UDI. The agent is based on the `java.lang.instrument` package introduced in JDK 1.5, ensuring portability. Figure 1b shows how application classes (`App`) are instrumented at load-time such that their instrumented versions (`INST_App`) are those actually linked by the JVM.

The current implementation of FERRARI uses Apache's bytecode engineering library BCEL [13]. UDIs are nevertheless free to use any bytecode engineering library, such as BCEL, ASM [18], Javassist [10], Soot [24], etc.

FERRARI offers generic mechanisms to ensure complete instrumentation coverage. To this end, (1) it ensures that UDI-inserted code is not executed before the JVM has completed bootstrapping and (2) it provides support for temporarily bypassing the execution of inserted code for each thread, e.g., during load-time instrumentation.

Regarding issue (1), FERRARI keeps a copy of the original code of every instrumented method and uses a global flag to bypass the execution of the UDI-inserted code during JVM boostrapping. That is, FERRARI uses code duplication within method bodies so as to branch to the original code during bootstrapping.

Concerning issue (2), FERRARI introduces a thread-local flag, the *Dynamic Inserted-code Bypass (DIB)*, which allows per-thread bypassing of the UDI-inserted code. To this end, FERRARI inserts the boolean instance field `dibFlag` into the `java.lang.Thread` class. If the `dibFlag` is set to `true`, the current thread's DIB is activated and the thread will bypass UDI-inserted code in invoked methods. The DIB is exposed to the UDI developer.

In general, methods invoked by UDI-inserted code must not themselves execute any UDI-inserted code, which otherwise could result in infinite recursions. Notably, UDI-inserted code often introduces dependencies on UDI-specific runtime classes. The UDI developer must ensure that methods in these classes do not execute any UDI-inserted code. To this end, the DIB can be used to temporarily bypass UDI-inserted code at runtime.

FERRARI's instrumentation agent, the UDI, UDI-specific runtime classes, and bytecode engineering library classes are excluded from normal instrumentation. All other classes are instrumented either statically or dynamically at load-time. For details, we refer to [5].

3. Completeness of CCTs

Below we define our criteria for completeness of CCTs; CCCP conforms to these criteria. In this paper, we are only concerned with completeness of the CCT structure; we are not considering properties of the dynamic metrics collected within CCT nodes, since they depend on the concrete application of CCCP.

We require that a *complete* CCT represents all method invocations where the caller or the callee is a Java method[1], after an initial JVM bootstrapping phase. This definition implies that in the following three situations, a method invocation must be represented in the CCT: (1) Java method invoked by another Java method; (2) native method invoked by a Java method; (3) Java method invoked by native code (e.g., callback from native code into bytecode through JNI, the Java Native Interface). We do not require an invocation of a native method by native code to be represented in the CCT, since conceptually the execution remains in native code (even though the native method may be invoked through JNI). Note that in general, native code execution cannot be profiled in a platform-independent way; in particular, native code execution is not amenable to any bytecode instrumentation technique.[2]

Since executing arbitrary inserted code during JVM bootstrapping may crash the JVM, calling context profiling during the bootstrapping phase is not possible without resorting to native code or to a modified JVM. Hence, we allow for an initial bootstrapping phase that is not reflected in the CCT. However, we require that the bootstrapping phase is over before the application's `main(String[])` method is called, and that the invocation of the `main(String[])` method is represented in the CCT. That is, the execution of Java application code is guaranteed to be completely represented in the CCT. After bootstrapping, Java methods invoked by system threads are also represented in the CCT. Furthermore, we require that the CCT does not include any spurious nodes due to CCCP; i.e., methods invoked by load-time instrumentation or by UDI-inserted code must not be represented in the CCT.

Regarding method invocation through reflection, the above definition of completeness implies that the called method of the reflection API (`Method.invoke(...)`, `Constructor.newInstance(...)`) must be represented in the CCT.

4. Complete Calling Context Profiling (CCCP)

In this section we present CCCP, which relies on instrumentation techniques and leverages FERRARI for comprehensive bytecode coverage.

In the remainder of this paper, we assume that each thread creates a separate, thread-confined CCT. Because of thread-confinement, we can avoid expensive synchronization upon CCT access. If a single CCT is desired, the thread-confined CCTs can be aggregated into a single structure, e.g., periodically by each thread, or after termination of a thread [6]. Alternatively, it is straightforward to adapt our approach to use a single, shared CCT root, which however requires atomic CCT operations, causing extra overhead.

In the following four subsections, we firstly discuss a simple approach that keeps the current CCT node in a thread-local variable; unfortunately, it is not practical because of high overhead. Secondly, we introduce the principles of CCCP, which is more efficient since it relies on passing the caller's CCT node to the callee as a special method argument. Thirdly, we explain the difficulties of CCCP, due to native code, reflection, and stack introspection.

[2] A feature of the JVM Tool Interface (JVMTI), "native method prefixing", allows wrapping native methods with Java methods that can be instrumented. However, this feature does not allow the instrumentation of native code. Furthermore, native code is needed to enable this feature before the first class is loaded during bootstrapping.

```
public static final ThreadLocal<CC> TL = new ThreadLocal<CC>() {
   protected CC initialValue() {
      return new CC(); // new CCT root for current thread
   }
};

void f() {
   CC caller = TL.get();
   CC callee = caller.profileCall("f()");
   TL.set(callee);
   try {
      ... // body of f()
   }
   finally { TL.set(caller); }
}
```

Figure 2. Simple but inefficient transformation of method `f()` for calling context profiling

Finally, we come up with a solution that reconciles completeness, accuracy, portability, and reduced overhead.

4.1 Accessing the CCT Node through a Thread-local Variable

One simple approach to calling context profiling is to keep the current CCT node in a thread-local variable `TL` that is updated upon method entry and method completion (see Figure 2). In this paper we assume that a CCT node is represented by an instance of type `CC` that offers (amongst others) the method `CC profileCall(String calleeID)`. It takes as argument a string uniquely identifying the callee method (e.g., fully qualified class name, method name, and method signature) and returns the `CC` instance representing the callee in the CCT.

However, this simple approach has two major drawbacks: Firstly, as native methods cannot be instrumented, their invocations will not be represented in the CCT, which will be incomplete. Secondly, accessing and updating `TL` upon method entry and method completion causes high overhead. We measured overhead up to factor 80 for the SPEC JVM98 [22] and DaCapo [8] benchmarks.

4.2 Direct Access to the CCT Node on the Call Stack

Below we outline the principles of CCCP, addressing the two problems of the simple approach discussed before.

Conceptually, the signatures of all methods are extended with one special argument to pass the caller's CCT node to the callee. Instead of maintaining a reference to the current CCT node in a single thread-local variable, each method invocation keeps a reference to its corresponding CCT node in a local variable on the call stack. Hence, the overhead of updating a thread-local variable is avoided. According to this idea, a non-native method `f()` gets transformed into method `f(CC caller)`, which we call *Extended-Signature-Instrumented (ESI)* method.

For compatibility with native code, it is essential to provide methods with the unmodified signature such that native code can invoke any Java method (through JNI, the Java Native Interface) without passing the special argument. We call these introduced methods with the original signature *Native-to-Bytecode (N2B)* wrappers. N2B wrappers also enable method invocation through the reflection API without passing the special argument.

Furthermore, because the signature of native methods cannot be modified, we insert methods that drop the special argument before invoking the corresponding native method; we call them *Bytecode-to-Native (B2N)* wrappers. This ensures that instrumented code may invoke any method (whether it is native or not) with the special argument.

Native methods and N2B wrappers preserve the original method signature, whereas ESI methods and B2N wrappers take a `CC` in-

stance as special argument. That is, conceptually we are *overloading* all methods, including also method declarations in interfaces. In order to prevent signature clashes, we assume that the type `CC` does not occur in the original program.

In our previous work [6], the presence of native code introduced some incompleteness and inaccuracy in the CCTs. First, CCT nodes representing native methods were not created. Second, a B2N wrapper simply discarded the special argument, and an N2B wrapper retrieved the CCT root (stored in a thread-local variable) to be passed to the ESI method. Consequently, if a Java method m_1 called native code that subsequently called back another Java method m_2, the resulting CCT was inaccurate, because the CCT node representing the call to m_2 appeared as a child of the CCT root (i.e., loss of calling context).

To solve these issues, in CCCP, a B2N wrapper creates the CCT node for the native method and preserves it in a thread-local variable during native code execution. That is, a B2N wrapper stores the CCT node before calling the native method and recovers the previously stored CCT node upon completion of the native method. Conversely, a N2B wrapper loads the CCT node from the thread-local variable.

4.3 Non-overloadable Methods

Ideally, all methods should be overloaded in the same way. Unfortunately, in current JDKs there are a few methods that cannot be overloaded; we call them *non-overloadable* methods. Non-overloadable methods are not necessarily excluded from instrumentation; normally, the bodies of non-overloadable Java methods can be instrumented.

We define our own notion of "virtual" and "nonvirtual" methods, which we need to distinguish in the following. The set of non-virtual methods includes static and private methods, as well as constructors; the set of virtual methods consists of all other methods.

A few JDK methods cannot be overloaded for three different reasons: (1) to work around a bug in Sun's recent JDKs that prevents the insertion of more than two virtual, non-final methods into `java.lang.Object`[3], (2) as an optimization to prevent the instrumentation of the `finalize()` method in `java.lang.Object`, and (3) to avoid interference with stack introspection performed by the JVM.

Concerning case (1), only `toString()` and `equals(Object)` in `java.lang.Object` are concerned.

With respect to case (2), we consider `Object.finalize()` as non-overloadable and also exclude it from instrumentation, since an instrumentation of that method would significantly increase the cost of garbage collection for each object. This is the only non-native method excluded from instrumentation.

Regarding case (3), invocations of wrapper methods (B2N or N2B) constitute extra frames on the call stack. Such additional stack frames may break native code in the virtual machine that performs stack introspection. For instance, in Sun's JDKs there are certain methods that rely on a fixed invocation sequence. Examples include methods in `java.lang.Class`, `java.lang.ClassLoader`, `java.lang.Runtime`, and `java.lang.System`. The execution of the affected methods requires inspection of the n topmost stack frames of callers to determine whether an operation shall be permitted; for each affected method, n is a statically known constant. If wrapper methods are added to the JDK, the additional stack frames due to the invocation of wrapper methods may violate the assumptions of the JDK programmer concerning the call stack. Those JDK methods that must not be invoked through an N2B or B2N wrapper are called *non-wrappable* methods in the following.

[3] `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6583051`

We need to make sure that there are no extra stack frames when invoking a non-wrappable method. For a non-wrappable Java method, this can be achieved by using code duplication instead of wrapping for compatibility with native code. That is, instead of introducing an N2B wrapper that calls the corresponding ESI method with the special argument, we can duplicate the instrumented method body in the method with the unmodified signature; we call the resulting method an *Original-Signature-Instrumented (OSI)* method.

However, as we cannot duplicate and adapt native code, this technique is not applicable for replacing B2N wrappers. Therefore, non-wrappable native methods are non-overloadable. For this reason, in Sun's JDKs, the following four native methods are non-overloadable:

```
java.lang.Class.forName0(String, boolean, ClassLoader)
java.sql.DriverManager.getCallerClassLoader()
java.util.ResourceBundle.getClassContext()
sun.reflect.Reflection.getCallerClass(int)
```

Code for stack introspection may also verify the signatures of methods on the call stack. Since ESI methods and B2N wrappers have extended signatures, their presence on the call stack may break such checks. In Sun's recent JDKs, the method `java.lang.reflect.Method.invoke(...)` is non-overloadable for this reason.

Because non-overloadable methods must be invoked without the special argument, it is necessary to consider polymorphic call sites and inheritance of non-overloadable methods. For each call site, the special argument must not be passed unless it can be statically guaranteed that the callee method is overloadable. Assume that the virtual method m_C in class C is non-overloadable. We analyze the class hierarchy in three consecutive steps in order to determine the methods that must be non-overloadable because of m_C; FERRARI includes a corresponding analysis tool.

1. For each (direct and indirect) superclass S of C, if S supports calls to a virtual method m_S with the same name and signature as m_C, then m_S is non-overloadable (m_S need not necessarily have an implementation in S; m_S may also be abstract, native, or inherited).
2. If a non-overloadable method is inherited to a subclass, the inherited method in the subclass is non-overloadable.
3. If a non-overloadable method m_X in class X can be invoked through an interface I that is a supertype of X, then method m_I in I with the same name and signature as m_X is non-overloadable.

4.4 Instrumentation Rules for CCCP

In Figure 3 and Figure 4 we present our implemented instrumentation rules that efficiently generate complete and accurate CCTs while dealing with non-overloadable and non-wrappable methods. Whenever possible, the caller's CCT node is passed as special argument; only upon invocation of a non-overloadable method, it is preserved in a thread-local variable.

In Figure 3 and Figure 4, certain combinations of the attributes (non-)native, (non-)overloadable, respectively (non-)wrappable do not exist, because the following two implications hold in general: (1) If a method is non-overloadable, it is also non-wrappable. (2) If a native method is non-wrappable, it is also non-overloadable.

We presume that the sets of non-overloadable respectively non-wrappable methods are statically known. In Sun's recent JDKs, there are about 50 non-wrappable Java methods where OSI methods have to be introduced instead of N2B wrappers. If the set of non-wrappable Java methods is not known, it can be conservatively assumed that all Java methods in the Java class library are non-

f() is non-native, overloadable, and wrappable:

```
void f() {            // N2B
   CC cc = TL.get();
   f(cc);
}
void f(CC cc) {       // ESI
   cc = cc.profileCall("f()");
   ... // instrumented body
}
```

f() is native, overloadable, and wrappable:

```
native void f();
void f(CC cc) {       // B2N
   cc = cc.profileCall("f()");
   CC old = TL.get();
   TL.set(cc);
   try { f(); }
   finally { TL.set(old); }
}
```

f() is non-native, overloadable, and non-wrappable:

```
void f() {            // OSI
   CC cc = TL.get();
   cc = cc.profileCall("f()");
   ... // instrumented body
}
void f(CC cc) {       // ESI
   cc = cc.profileCall("f()");
   ... // instrumented body
}
```

f() is non-native, non-overloadable, and non-wrappable:

```
void f() {            // OSI
   CC cc = TL.get();
   cc = cc.profileCall("f()");
   ... // instrumented body
}
```

f() is native, non-overloadable, and non-wrappable:

```
native void f();
```

Figure 3. Method overloading and instrumentation of method entries

f() is overloadable:

```
...
f(cc);
...
```

f() is non-native and non-overloadable:

```
...
CC old = TL.get();
TL.set(cc);
try { f(); }
finally { TL.set(old); }
...
```

f() is native and non-overloadable:

```
...
CC cc_f = cc.profileCall("f()");
CC old = TL.get();
TL.set(cc_f);
try { f(); }
finally { TL.set(old); }
...
```

Figure 4. Instrumentation of an invocation of `f()`. The variable `cc` refers to the current CCT node.

wrappable. Consequently, OSI methods would always be created instead of N2B wrappers, which however causes significant code bloat.

We also assume that all native non-overloadable methods are invoked only from monomorphic call-sites such that the caller can profile the invocation of the native callee. For Sun's JDKs, this assumption holds (tested with Sun JDK 1.5, 1.6, and 1.7 using different operating systems, as well as with various ports of Sun's JDKs); in particular, all native non-overloadable methods are nonvirtual.[4]

5. Use Cases

In this section we briefly discuss two use cases, platform-independent profiling and cross-profiling for embedded Java processors. Both use cases benefit from CCCP.

5.1 Platform-independent Profiling (JP-UDI)

In [6], we introduced platform-independent calling context profiling. In this approach, each node in the CCT stores only platform-independent metrics [14], such as the number of method invocations (with the same call stack), the number of bytecodes executed in the calling context, as well as the number of allocated objects of various types (including array allocation). Platform-independent profiling does not aim at measuring CPU consumption or memory consumption on the heap, since these metrics largely depend on the profiling environment (hardware, operating system, and virtual machine) and are subject to significant measurement perturbations caused by the profiling.

If the profiled application is deterministic, platform-independent profiles are reproducible in different environments, as long as the same Java class library is used. In practice, some level of non-determinism is usually unavoidable; e.g., identity hash codes may differ, thread scheduling may be different, etc. Nonetheless, in [6] we have shown that the overlap percentage [3] of platform-independent profiles generated in different environments for the same workload can exceed 99% for some standard benchmarks.

In our prior work on calling context profiling [6], the generated CCTs were incomplete and inaccurate in the presence of native code. Hence, we implemented CCCP in the JP-UDI, superseding our prior tools for platform-independent profiling. All measurements presented in this paper are based on the new JP-UDI.

The JP-UDI supports user-defined profilers to process the collected profiling data in regular intervals. The profiler's execution does not create any artifacts in the generated profiles, because the

[4] If this assumption does not hold in the JDK of a different vendor, we could not create the callee's CCT node within the caller (because the callee may not be statically known in the case of dynamic binding). In that case, invocations of native non-overloadable methods would not be represented in the CCT.

DIB (see Section 2) is activated before invoking the profiler. Hence, the profiler developer can use the whole Java class library without affecting the profiles (apart from possible perturbations due to different thread scheduling). We are using a profiler that aggregates the per-thread CCTs into a single CCT upon program termination with the aid of a JVM shutdown hook.

5.2 Cross-profiling (CProf-UDI)

Cross-profiling allows collecting profiling data on a *host* platform using the execution time metric of a *target* platform [12, 7]. Embedded systems with limited computing resources are particularly interesting cross-profiling targets, since they often do not provide proper profiling support. Generating a CCT on an embedded system may be impossible because of memory constraints; the CCT may consume significantly more heap memory than the embedded application being profiled. Cross-profiling helps developers analyze their applications without prior deployment on the target system. As in the case of platform-independent profiling, the completeness and accuracy of the CCTs generated by CCCP advances the quality of cross-profiling with respect to our prior work [7].

The CProf-UDI enables calling context cross-profiling for embedded Java systems where accurate cycle estimates are available for most bytecodes. In particular, some recent embedded Java processors meet this criterion. A Java processor implements the JVM in hardware; its instruction set corresponds to the JVM bytecodes.

In order to estimate the CPU cycle consumption on the target, we maintain a *cycle counter* in each CCT node, which is updated in the beginning of each basic block of code according to the sum of the cycle estimates for all bytecodes in the basic block, disregarding method invocation and return bytecodes, which are treated specially. The cycle estimate for each method invocation respectively return bytecode is computed dynamically at runtime, since it may depend on the size of the callee method (for invocation) respectively caller method (for return).

Similar to the use case described in Section 5.1, we aggregate the individual CCTs of all threads into a single CCT upon program termination. However, we exclude the CCTs of system threads in the host JVM, since these threads do not execute on the target processor; taking the CCTs of system threads into account would reduce the accuracy of the generated cross-profile.

In order to confirm the feasibility of cross-profiling, we configured the CProf-UDI to use CPU cycle estimates for the Java Optimized Processor JOP [19]. We assessed the accuracy of the generated cross-profiles with the embedded benchmarks JavaBenchEmbedded[5] (JBE).

We executed them on the JOP hardware to obtain the exact cycle consumption as reference. Afterwards, we collected the corresponding cross-profiles using Sun JDK 1.7 on a Linux Fedora Core 2 machine, aggregated the CPU cycles of all calling contexts, and computed the percent error. For all measured benchmarks and settings, the errors are below 3.3%. The errors are due to the use of different Java class libraries on the host and on the target, as well as due to imprecise cycle estimates for some bytecodes. While the generated CCTs are structurally complete and accurate, the cycle estimates within the CCT nodes may suffer from some inaccuracies.

6. Evaluation

In this section, (1) we analyze the CCTs produced by CCCP, (2) we use the generated CCTs to explore the dynamic behavior of typical Java workloads regarding their usage of the standard Java class library, and, (3) we assess the runtime overhead due to CCCP.

[5] `http://www.jopwiki.com/JavaBenchEmbedded`

	Native Methods in CCT				Overlap with
	# Nodes	(%)	# Calls	(%)	Incomplete CCT
DaCapo					
antlr	38209	(5.98 %)	3228460	(1.94 %)	96.20 %
bloat	45858	(6.74 %)	21811724	(1.87 %)	78.23 %
chart	5322	(6.86 %)	21000464	(4.43 %)	89.64 %
eclipse	57546	(2.77 %)	19925955	(2.04 %)	99.50 %
fop	4602	(3.78 %)	1394460	(2.91 %)	78.42 %
hsqldb	3137	(5.67 %)	267975	(0.12 %)	99.50 %
jython	122211	(5.95 %)	73774270	(7.54 %)	52.85 %
luindex	3418	(6.67 %)	4436959	(0.97 %)	96.81 %
lusearch	1425	(6.92 %)	30283467	(5.81 %)	94.05 %
pmd	5580	(0.69 %)	15399269	(3.11 %)	95.17 %
xalan	6248	(3.52 %)	4049835	(0.62 %)	97.26 %
JVM98					
compress	447	(7.49 %)	4749	(0.00 %)	99.99 %
jess	915	(5.18 %)	2669287	(2.24 %)	90.96 %
db	580	(6.26 %)	80415	(0.05 %)	87.39 %
javac	56071	(5.06 %)	4238182	(3.67 %)	86.35 %
mpegaudio	474	(6.77 %)	1248717	(1.14 %)	98.83 %
mtrt	492	(6.16 %)	195820	(0.07 %)	99.92 %
jack	3206	(5.57 %)	2991208	(4.29 %)	89.65 %

Table 1. Native methods represented in the CCTs and overlap percentage of the CCTs produced by CCCP with CCTs generated by former instrumentation rules that do not guarantee completeness and accuracy

Our measurements were obtained with the JP-UDI for platform-independent profiling (see Section 5.1).[6] We used two benchmark suites, SPEC JVM98 [22] (problem size 100) and DaCapo [8] (version 'dacapo-2006-10-MR2', default workload size). SPEC JVM98 consists of 7 benchmarks, whereas DaCapo, which is more recent and is intended to provide more realistic workloads than SPEC JVM98, consists of 11 benchmarks. Our test platform was a Linux Fedora Core 2 computer (Intel Pentium 4, 2.66 GHz, 1024 MB RAM). We removed background processes as much as possible in order to obtain reproducible results. All measurements were obtained with Sun JDK 1.7.0-ea-b21.

6.1 CCT Analysis

In contrast to our prior work [6], CCCP ensures complete and accurate CCTs. In order to quantify these benefits, we explore the contribution of native method invocations to the overall CCTs. Native method calls were not represented in the CCTs produced by the instrumentation rules in [6]. In Table 1, the columns "Native Methods" show the number of CCT nodes corresponding to native methods ("# Nodes") and the number of invocations to these native methods ("# Calls"). The percentages in braces are relative to the total number of CCT nodes, respectively to the total number of method calls. For 'jython', native methods contribute more than 7.5% to the overall method calls. Therefore, it is important to represent native method calls in the CCTs.

In Table 1, we also show the *overlap percentage* [3] of CCTs produced by CCCP and CCTs generated by the instrumentation rules in [6]. Informally, the overlap represents the percentage of profiled information weighted by execution frequency that exists in both CCTs. Two identical CCTs have an overlap percentage of 100%. We define the overlap percentage of two CCTs as follows:

[6] The CProf-UDI for cross-profiling presented in Section 5.2 causes higher overhead than the JP-UDI, if the cycle estimates for invocation and return bytecodes are computed at runtime. If constant cycle estimates were used for these bytecodes, the cross-profiling overhead would correspond to the measurements presented here.

	App.→App.	JDK→App.	App.→JDK	JDK→JDK
DaCapo				
antlr	58.33 %	0.10 %	5.49 %	36.08 %
bloat	32.16 %	14.91 %	18.06 %	34.87 %
chart	39.97 %	1.31 %	22.11 %	36.61 %
eclipse	62.31 %	0.18 %	9.11 %	28.40 %
fop	33.61 %	0.08 %	16.62 %	49.69 %
hsqldb	84.07 %	0.01 %	12.51 %	3.41 %
jython	41.43 %	0.70 %	7.28 %	50.59 %
luindex	79.85 %	0.44 %	7.50 %	12.21 %
lusearch	70.96 %	0.00 %	9.30 %	19.74 %
pmd	59.87 %	1.57 %	14.59 %	23.97 %
xalan	43.57 %	0.17 %	11.23 %	45.03 %
JVM98				
compress	99.96 %	0.00 %	0.00 %	0.04 %
jess	82.69 %	0.04 %	9.70 %	7.57 %
db	0.08 %	1.26 %	74.68 %	23.98 %
javac	51.52 %	0.55 %	19.87 %	28.06 %
mpegaudio	99.78 %	0.02 %	0.12 %	0.08 %
mrtr	99.14 %	0.00 %	0.61 %	0.25 %
jack	11.31 %	0.96 %	33.48 %	54.25 %

Table 2. Intra/inter application/JDK method calls

- We regard a CCT X as a set of calling contexts C_{X_i}. Each C_{X_i} is identified by its call stack, $stack(C_{X_i})$, and has an associated number of calls, $calls(C_{X_i})$.
- $rc(C_{X_i}, X)$ is the relative number of calls in C_{X_i}:

$$rc(C_{X_i}, X) = \frac{calls(C_{X_i})}{\sum_{C_{X_k} \in X} calls(C_{X_k})}$$

- The overlap percentage $overlap(A,B)$ of two CCTs A and B is computed as follows:

$$overlap(A,B) = \sum_{\substack{C_{A_i} \in A,\\ C_{B_j} \in B,\\ stack(C_{A_i}) = stack(C_{B_j})}} \min(rc(C_{A_i}, A), rc(C_{B_j}, B))$$

For some benchmarks, the overlap percentage is surprisingly low. For example, for 'jython', the overlap percentage is only 52.85%. The reason is not only the incompleteness of prior CCTs concerning native methods, but also their inaccuracy. In [6], callbacks from native code into bytecode loose the calling context; the invoked Java methods are incorrectly represented as children nodes of the CCT root. Hence, whole subtrees are misplaced in the resulting CCT, significantly reducing the overlap with an accurate CCT.

We conclude that for realistic Java workloads, the gains in CCT completeness and accuracy achieved by CCCP are important for analyzing dynamic program behavior.

6.2 Analysis of the Usage of the Java Class Library

In this subsection we leverage the complete and accurate CCTs produced by CCCP in order to explore how our benchmarks make use of the standard Java class library.

Table 2 shows the inter/intra application/JDK call percentages. This kind of information is valuable for a better understanding of the dynamic behavior of Java applications. Table 2 allows us to differentiate between "application-intensive" and "JDK-intensive" benchmarks. For example, 'compress', 'mpegaudio', and 'mtrt' are application-intensive, whereas 'db' is JDK-intensive. The same conclusion can be drawn from the data published in [17], although the exact measurements are different from ours, because of distinct Java class libraries. However, our measurements allow a more detailed analysis, because we can distinguish between calls originating from the application and calls originating from the JDK.

In Table 2, 'bloat' exhibits a relatively high invocation percentage from JDK to application code. An analysis of the CCT revealed that this corresponds to invocations from the JDK to overridden `toString()` methods in application code when performing string buffer manipulations. Moreover, we observe that 'compress' reports 0.04% of intra JDK invocations, while there seem to be no invocations from application to JDK code. Our analysis showed that there were actually 0.0031% such invocations, notably to `java.io` for reading benchmark input data, explaining the rounded 0.00%.

Prevailing approaches for analyzing the dynamic behavior of Java systems resort to a modified JVM or to native code. For instance, in [8], the authors employed a modified version of the Jikes RVM [1] to gather static and dynamic metrics for SPEC JVM98 and DaCapo. Similarly, in [17], the authors used a modified Kaffe Virtual Machine [26]. In contrast, our approach enables such analyses with unmodified, standard, state-of-the-art JVMs.

6.3 Performance Evaluation

Since our instrumentation rules are not specific to the JP-UDI, but can be used for implementing any Java tool that requires complete calling context information, it is important to understand the different sources of overhead. Hence, we first provide a detailed analysis to differentiate between the overhead due to FERRARI's instrumentation (i.e., the creation of bypasses that cause some code bloat) and the overhead due to calling context profiling (implemented by the JP-UDI). Second, as dynamic bytecode instrumentation at load-time increases the overall execution time, we analyze its impact and describe an optimization to reduce FERRARI's instrumentation overhead.

For the performance evaluation, we used two different just-in-time compilers, the HotSpot Client VM ("client mode") and the HotSpot Server VM ("server mode").

6.3.1 Overhead of FERRARI with JP-UDI versus FERRARI without UDI

In the following we analyze the sources of overhead, considering the FERRARI framework and the JP-UDI.

Firstly, we use FERRARI with the JP-UDI to run the benchmarks. These measurements give us the overall overhead caused by FERRARI and by our instrumentation for calling context profiling. Secondly, we measure the overhead of FERRARI alone (i.e., without the perturbations due to the code inserted by the JP-UDI). To this end, we let FERRARI create the code for bypasses in all methods and run the benchmarks with the `dibFlag` set to `true` for all threads during the entire execution. By computing the difference between the overhead of FERRARI with and without the JP-UDI, we can estimate the overhead of CCT creation.

The FERRARI-proper overhead stems from two sources: (1) inserted conditionals (checking the bootstrapping bypass flag and the `dibFlag`), and, (2) code bloat (keeping a copy of the original method body together with the instrumented version). Our measurements aim at separating these two sources of framework overhead.

Table 3 and Table 4 present the execution times for the various settings and the corresponding overhead factors ("ovh"). The column "Orig." gives the execution time of the original, unmodified benchmarks. The rows "Geo. mean" represent the geometric means for the SPEC JVM98 respectively the DaCapo suite.

To quantify the impact of code bloat, besides the normal "With Bloat" setting, we introduce a dummy "Avoid Bloat" instrumentation mode, where only the conditionals for the bypasses are inserted, but no UDI-instrumented code. To this end, methods added

	Orig.	Only FERRARI				FERRARI and JP-UDI	
		Avoid Bloat		With Bloat			
DaCapo	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
antlr	3.31	4.28	**1.29**	4.44	**1.34**	16.50	**4.98**
bloat	13.79	16.37	**1.19**	16.98	**1.23**	79.90	**5.79**
chart	11.35	12.86	**1.13**	12.94	**1.14**	67.20	**5.92**
eclipse	59.44	62.92	**1.06**	65.57	**1.10**	222.62	**3.75**
fop	2.15	2.40	**1.12**	2.44	**1.13**	5.57	**2.59**
hsqldb	7.59	7.80	**1.03**	7.97	**1.05**	21.16	**2.79**
jython	9.15	11.40	**1.25**	11.50	**1.26**	42.20	**4.61**
luindex	15.94	17.39	**1.09**	18.05	**1.13**	83.28	**5.22**
lusearch	18.84	21.09	**1.12**	21.61	**1.15**	64.03	**3.40**
pmd	11.95	25.63	**2.14**	26.64	**2.23**	39.80	**3.33**
xalan	18.82	24.25	**1.29**	25.55	**1.36**	95.60	**5.08**
Geo.mean	11.09	13.52	**1.22**	13.92	**1.26**	46.08	**4.16**
JVM98	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.73	8.49	**1.48**	9.15	**1.60**	20.38	**3.56**
jess	1.46	2.33	**1.60**	2.36	**1.62**	8.57	**5.87**
db	14.14	14.57	**1.03**	15.12	**1.07**	24.29	**1.72**
javac	3.95	4.90	**1.24**	4.98	**1.26**	16.24	**4.11**
mpegaudio	2.47	3.78	**1.53**	3.79	**1.53**	9.72	**3.94**
mtrt	1.15	2.30	**2.00**	2.34	**2.03**	15.54	**13.51**
jack	3.48	4.19	**1.20**	4.22	**1.21**	10.87	**3.12**
Geo.mean	3.34	4.71	**1.41**	4.82	**1.45**	14.14	**4.24**

Table 3. Overhead of FERRARI versus FERRARI and JP-UDI; *client mode*

	Orig.	Only FERRARI				FERRARI and JP-UDI	
		Avoid Bloat		With Bloat			
DaCapo	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
antlr	3.80	4.00	**1.05**	4.13	**1.09**	12.86	**3.38**
bloat	8.62	9.42	**1.09**	9.53	**1.11**	50.34	**5.84**
chart	9.20	9.36	**1.02**	9.46	**1.03**	26.84	**2.92**
eclipse	47.30	50.47	**1.07**	50.78	**1.07**	205.38	**4.34**
fop	2.01	2.12	**1.05**	2.21	**1.10**	5.93	**2.95**
hsqldb	5.13	5.75	**1.12**	5.79	**1.13**	15.04	**2.93**
jython	5.66	7.62	**1.35**	7.81	**1.38**	27.04	**4.78**
luindex	13.12	13.13	**1.00**	13.27	**1.01**	79.64	**6.07**
lusearch	13.41	13.45	**1.00**	13.64	**1.00**	73.90	**5.51**
pmd	9.86	10.47	**1.06**	10.69	**1.08**	29.26	**2.97**
xalan	14.85	17.89	**1.20**	18.12	**1.22**	87.69	**5.91**
Geo.mean	8.69	9.47	**1.09**	9.63	**1.11**	35.96	**4.14**
JVM98	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	5.68	6.68	**1.18**	6.69	**1.18**	14.69	**2.59**
jess	1.47	1.52	**1.03**	1.63	**1.11**	6.16	**4.19**
db	13.71	13.83	**1.01**	13.88	**1.01**	18.22	**1.33**
javac	3.79	4.06	**1.07**	4.07	**1.07**	15.54	**4.10**
mpegaudio	2.48	2.51	**1.01**	2.52	**1.02**	7.15	**2.88**
mtrt	1.16	1.20	**1.03**	1.33	**1.15**	11.59	**9.99**
jack	3.48	3.51	**1.01**	3.57	**1.03**	8.20	**2.36**
Geo.mean	3.31	3.47	**1.05**	3.57	**1.08**	10.82	**3.27**

Table 4. Overhead of FERRARI versus FERRARI and JP-UDI; *server mode*

by the UDI are discarded, and for changed methods, the UDI-instrumented body is replaced with a short bytecode sequence that throws `UnsupportedOperationException`. For this to work, the `dibFlag` must remain `true` throughout the execution, so as to never branch to any (non-existing) UDI-instrumented code.

Table 3 and Table 4 show the overhead for normal execution ("FERRARI and JP-UDI") and for the two settings where all UDI-inserted code is bypassed ("Only FERRARI"; "Avoid Bloat" respectively "With Bloat"). The measurements represent the median of 15 runs within the same JVM process. Hence, the impact of load-time instrumentation is relatively insignificant. We evaluate the impact of dynamic instrumentation in Section 6.3.2.

For the "FERRARI and JP-UDI" setting, the overhead is relatively moderate, considering that the instrumentation covers both application and JDK classes, creates complete CCTs, and correctly handles transitions between bytecode and native code. In client mode, on average the overhead is a factor of 4.16–4.24; in server mode, the average overhead is only a factor of 3.27–4.14.

Comparing the "FERRARI and JP-UDI" and "Only FERRARI With Bloat" settings, we can see that the difference in overhead, i.e., the overhead of the instrumentation performed by the JP-UDI, is a factor of 0.65–11.48 in client mode, respectively a factor of 0.32–8.84 in server mode.

For the 'mtrt' benchmark, we experience the highest overhead of factor 13.51 in client mode respectively factor 9.99 in server mode. 'mtrt' is the most object-oriented benchmark in the SPEC JVM98 suite according to [15], i.e., it involves many invocations of small methods, which explains the higher overhead.

For the setting "Only FERRARI With Bloat", the overhead is a factor of 1.05–2.23 in client mode, respectively a factor of 1.01–1.38 in server mode. Comparing the "Avoid Bloat" and "With Bloat" settings, the difference in overhead is very low (0–12%). We conclude that the code bloat does not significantly hinder code optimizations by the just-in-time compiler.

6.3.2 Overhead of Dynamic Load-time Instrumentation

In the following we consider the performance impact of dynamic load-time instrumentation, which contributes to the overall program execution time. We compare it with a setting where the benchmark classes are statically instrumented before execution. We also measure an optimization to reduce the dynamic instrumentation overhead.

As dynamic instrumentation essentially takes place in the startup phase, we set the parameters of the benchmark suites to run a single benchmark exactly once in a separate JVM process. We repeated these measurements 15 times, noting execution times as reported by the operating system, and show here the median of these values.

Table 5 and Table 6 present the execution times for the various settings. Note that the overheads presented in Tables 5 and 6 are not directly comparable with the those given in Tables 3 and 4, since the reference measurements ("Orig.") are different.

The column "Dynamic" shows the measurements with dynamic load-time instrumentation of application classes. Class files are communicated as byte arrays between FERRARI's instrumentation agent and the JP-UDI (see Figure 1 on page 2). The observed overhead is of factor 1.81–9.9 in client mode, respectively of factor 1.65–9.0 in server mode.

The column "Dyn. opt." presents the results of an optimization, exploiting the fact that both FERRARI and the JP-UDI use BCEL [13] for code transformations. FERRARI offers an optimized interface for UDIs that are implemented with BCEL. Instead of a byte array, FERRARI passes an instance of the BCEL abstraction `org.apache.bcel.generic.ClassGen` to the UDI for manipulation. FERRARI keeps a deep copy of the unmodified parsed class bytes. This optimization avoids unnecessary parsing and dumping of the class bytes and reduces the overhead by a factor of 0.01–0.38 in client mode, respectively by a factor of 0.02–0.36 in server mode.

The column "Static" corresponds to a setting where all benchmark classes are statically instrumented. Although FERRARI and the JP-UDI have not been designed for this setting, it provides useful reference measurements for comparison with load-time instrumentation. For this experiment, FERRARI's dynamic instrumentation agent was replaced with one that only signals the end of bootstrapping but does not perform any instrumentation. Be-

	Orig.	Dynamic		Dyn. opt.		Static	
DaCapo	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**		
antlr	12.49	40.57	**3.25**	39.96	**3.20**		
bloat	21.38	88.28	**4.13**	87.51	**4.09**		
chart	18.84	67.26	**3.57**	62.62	**3.32**		
eclipse	121.25	370.70	**3.06**	360.69	**2.97**		
fop	6.52	40.88	**6.27**	38.52	**5.91**		
hsqldb	11.09	45.05	**4.06**	41.96	**3.78**	**N/A**	
jython	34.71	134.89	**3.89**	124.52	**3.59**		
luindex	20.00	103.66	**5.18**	101.15	**5.06**		
lusearch	21.11	109.63	**5.19**	109.45	**5.18**		
pmd	23.03	77.84	**3.38**	70.17	**3.05**		
xalan	44.93	163.55	**3.64**	149.96	**3.34**		
Geo.mean	22.27	90.18	**4.05**	85.79	**3.85**		
JVM98	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	6.55	24.85	**3.79**	24.38	**3.72**	21.31	**3.25**
jess	2.75	14.36	**5.22**	13.31	**4.84**	10.02	**3.64**
db	16.27	29.46	**1.81**	28.90	**1.78**	27.25	**1.67**
javac	6.11	25.58	**4.19**	23.62	**3.87**	19.57	**3.20**
mpegaudio	3.37	14.07	**4.18**	13.73	**4.07**	11.07	**3.28**
mtrt	2.02	19.99	**9.90**	19.97	**9.89**	16.70	**8.27**
jack	4.61	16.27	**3.53**	15.77	**3.42**	12.77	**2.77**
Geo.mean	4.77	19.89	**4.17**	19.19	**4.02**	16.00	**3.35**

Table 5. Overhead of dynamic, dynamic-optimized, and static instrumentation; *client mode*

	Orig.	Dynamic		Dyn. opt.		Static	
DaCapo	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**		
antlr	12.40	40.58	**3.27**	40.19	**3.24**		
bloat	21.46	86.58	**4.03**	85.26	**3.97**		
chart	19.05	66.52	**3.49**	63.29	**3.32**		
eclipse	124.77	374.10	**3.00**	359.02	**2.88**		
fop	6.55	41.10	**6.27**	38.69	**5.91**		
hsqldb	11.01	44.96	**4.08**	42.94	**3.90**	**N/A**	
jython	35.41	134.26	**3.79**	123.77	**3.50**		
luindex	20.09	103.23	**5.14**	101.24	**5.04**		
lusearch	21.23	108.68	**5.12**	108.37	**5.10**		
pmd	23.24	77.97	**3.35**	74.86	**3.22**		
xalan	46.89	160.86	**3.43**	151.02	**3.22**		
Geo. Mean	22.51	89.77	**3.99**	86.33	**3.84**		
JVM98	**[s]**	**[s]**	**ovh**	**[s]**	**ovh**	**[s]**	**ovh**
compress	6.24	20.38	**3.27**	19.74	**3.16**	14.92	**2.39**
jess	5.97	20.06	**3.36**	18.29	**3.06**	11.46	**1.92**
db	15.76	25.93	**1.65**	25.22	**1.60**	21.78	**1.38**
javac	17.71	53.12	**3.00**	51.22	**2.89**	43.57	**2.46**
mpegaudio	3.96	17.40	**4.39**	17.08	**4.31**	10.62	**2.68**
mtrt	3.33	29.98	**9.00**	29.81	**8.95**	23.35	**7.01**
jack	10.05	23.40	**2.33**	22.58	**2.25**	15.60	**1.55**
Geo. Mean	7.53	25.43	**3.38**	24.54	**3.26**	18.06	**2.40**

Table 6. Overhead of dynamic, dynamic-optimized, and static instrumentation; *server mode*

cause CCCP overloads methods to pass the special argument, it is essential that every class is instrumented in the same way. Otherwise, a class excluded from instrumentation may fail to implement/override an added method in a supertype, or invocations with the special argument fail if the target method is not overloaded. Because the DaCapo suite involves dynamically generated classes that cannot be statically instrumented, we excluded DaCapo from this experiment.

For SPEC JVM98, subtracting the overhead of static instrumentation from the dynamic settings gives an estimation of the overhead due to load-time instrumentation, which is up to factor 1.63 in client mode and up to factor 1.99 in server mode. The extra overhead due to dynamic instrumentation is paid for mostly during the application startup phase. Consequently, for longer running applications, the difference in overhead between the static and dynamic settings will be smaller.

7. Related Work

In this section we discuss related work in two areas, calling context profiling and bytecode instrumentation.

7.1 Calling Context Profiling

Calling context profiles can be valuable for inter-procedural code optimization and overall program understanding. Calling context profiling has been explored by many researchers. Existing approaches that create accurate CCTs [20, 2] suffer from considerable overhead. Approaches based on sampling and stack-walking [25, 4] help reduce the overhead, but at the price of a loss of accuracy. In [27], a hybrid approach called "adaptive bursting" is proposed, which improves sampling-based stack-walking. It selectively disables bursts from previously sampled calling contexts, thus reducing redundant samples due to repetitive execution sequences. Unfortunately, this approach does not create complete CCTs and relies on native code (as it is based on the JVMPI [21]), hence limiting its portability. In contrast, CCCP reconciles both complete CCT creation and moderate overhead, while ensuring full portability.

In [9], the Probabilistic Calling Context (PCC) approach continuously maintains a probabilistically unique value representing the current calling context in a hash table. This approach causes rather low overhead (as the value corresponding to a calling context can be efficiently computed). However, PCC has three major drawbacks: Firstly, it is based on a modified Jikes RVM [1], thus preventing its use on standard JVMs. Secondly, PCC is not conflict-free by design, as several calling contexts can be mapped to the same value, thus reducing accuracy. Thirdly, PCC excludes methods in the Java class library from the calling context and does not accurately handle calling context when native code calls back into bytecode. PCC only instruments application methods. In contrast, thanks to FERRARI, CCCP instruments all methods so as to provide the complete and accurate calling context without resorting to any JVM modification.

7.2 Bytecode Instrumentation

There are many tools for manipulating Java bytecode, such as BCEL, ASM [18], Javassist [10], Soot [24], or JOIE [11], to cite some of them. FERRARI's static instrumentation tool and its dynamic instrumentation agent are based on BCEL. Nonetheless, the UDI developers are free to use the bytecode instrumentation library of their choice.

There are some techniques aiming at the application of user-defined instrumentations to the Java class library. The Twin Class Hierarchy (TCH) [16] replicates the full hierarchy of instrumented JDK classes into a separate package that coexists with the original one. This technique has the disadvantage that the applications need to be instrumented to refer explicitly to a given JDK version (original or instrumented). In addition, as pointed out in [23], the use of replicated classes limits the applicability of instrumentation in presence of native code, and therefore TCH cannot be used safely and transparently to instrument the complete Java class library.

FERRARI, in contrast, does not use class replication techniques, but rather code duplication within the method bodies, allowing instrumentation with full bytecode coverage independently of the presence of native methods. Our approach, in addition, ensures total transparency to the application (the application does not need to know if the Java class library is instrumented or not) and FERRARI's DIB mechanism allows switching between original and inserted code at runtime.

8. Conclusion

In this paper we presented CCCP, an approach to calling context profiling in Java that reconciles completeness, accuracy, portability, and reduced overhead. CCCP relies on FERRARI, a generic bytecode instrumentation framework which allows user-defined instrumentation modules to instrument all Java methods executing in a JVM.

CCCP yields complete and accurate CCTs, correctly handling transitions between bytecode and native code. CCCP reduces the runtime overhead by passing the calling context information as a special method argument whenever possible. It also deals with the intricacies of native code and stack introspection. Whereas other approaches to calling context profiling rely on native code or on a modified JVM to create accurate calling context information, CCCP is implemented in pure Java to ensures portability and compatibility with state-of-the-art JVMs.

Thanks to the accuracy and completeness of the generated CCTs, CCCP eases the analysis of the dynamic behavior of Java systems. In this paper, we analyzed how the SPEC JVM98 and DaCapo benchmarks make use of the Java class library. Moreover, we presented a detailed evaluation of CCCP and FERRARI, exploring the different sources of overhead. Using the SPEC JVM98 and DaCapo benchmark suites, we confirmed that CCCP is practical for typical Java workloads.

Acknowledgments

The work presented in this paper has been supported by the Swiss National Science Foundation.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, B. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, N. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

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

[3] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[4] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Research report, IBM, July 2000.

[5] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ 2007 (5th International Conference on Principles and Practices of Programming in Java)*, pages 135–144, Lisbon, Portugal, 2007. ACM Press.

[6] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 2008. `http://dx.doi.org/10.1002/spe.890`.

[7] W. Binder, A. Villazón, M. Schoeberl, and P. Moret. Cache-aware cross-profiling for Java processors. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES-2008)*, pages 127–136, Atlanta, Georgia, USA, Oct. 2008. ACM.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, Oct. 2006. ACM Press.

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

[10] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[11] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[12] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. The efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[13] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.

[14] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[15] J. Dujmovic and C. Herder. Visualization of Java workloads using ternary diagrams. *Software Engineering Notes*, 29(1):261–265, 2004.

[16] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 288–300, New York, NY, USA, 2004. ACM.

[17] D. Gregg, J. F. Power, and J. Waldron. A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 17(7–8):757–773, 2005.

[18] ObjectWeb. ASM. Web pages at `http://asm.objectweb.org/`.

[19] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[20] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[21] Sun Microsystems, Inc. Java Virtual Machine Profiler Interface (JVMPI). Web pages at `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/`, 2000.

[22] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. Web pages at `http://www.spec.org/osg/jvm98/`, 1998.

[23] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 89–94, New York, NY, USA, 2006. ACM.

[24] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[25] J. Whaley. A portable sampling-based profiler for Java Virtual Machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87. ACM Press, June 2000.

[26] T. Wilkinson. Kaffe - a Java virtual machine. Web pages at `http://www.kaffe.org/`.

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