A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods

Alexandre Courbot[1], Mariela Pavlova[2], Gilles Grimaud[1], and Jean-Jacques Vandewalle[3]

[1] IRCICA/LIFL, Univ. Lille 1, France, INRIA futurs, POPS Research Group
{Alexandre.Courbot, Gilles.Grimaud}@lifl.fr
[2] INRIA Sophia-Antipolis, France, Everest Research Group
Mariela.Pavlova@sophia.inria.fr
[3] Gemplus Systems Research Labs, La Ciotat, France
Jean-Jacques.Vandewalle@research.gemplus.com

Abstract. Ahead-of-Time and Just-in-Time compilation are common ways to improve runtime performances of restrained systems like Java Card by turning critical Java methods into native code. However, native code is much bigger than Java bytecode, which severely limits or even forbids these practices for devices with memory constraints.
In this paper, we describe and evaluate a method for reducing natively-compiled code by suppressing runtime exception check sites, which are emitted when compiling bytecodes that may potentially throw runtime exceptions. This is made possible by completing the Java program with JML annotations, and using a theorem prover in order to formally prove that the compiled methods never throw runtime exceptions. Runtime exception check sites can then safely be removed from the generated native code, as it is proved they will never be entered.
We have experimented our approach on several card-range and embedded Java applications, and were able to remove almost all the exception check sites. Results show memory footprints for native code that are up to 70% smaller than the non-optimized version, and sometimes as low than 115% the size of the Java bytecode when compiled for ARM thumb.

1 Introduction

Enabling Java on embedded and restrained systems is an important challenge for today's industry and research groups [1]. Java brings features like execution safety and low-footprint program code that make this technology appealing for embedded devices which have obvious memory restrictions, as the success of Java Card witnesses. However, the memory footprint and safety features of Java come at the price of a slower program execution, which can be a problem when the host device already has a limited processing power. As of today, the interest of Java for smart cards is still growing, with next generation operating systems for smart cards that are closer to standard Java systems [2, 3], but runtime performance in still an issue. To improve the runtime performance of Java systems, a common practice is to translate some parts of the program bytecode into native code.

Doing so removes the interpretation layer and improves the execution speed, but also greatly increases the memory footprint of the program: it is expected that native code is about three to four times the size of its Java counterpart, depending on the target architecture. This is explained by the less-compact form of native instructions, but also by the fact that many safety-checks that are implemented by the virtual machine must be reproduced in the native code. For instance, before dereferencing a pointer, the virtual machine checks whether it is `null` and, if it is, throws a `NullPointerException`. Every time a bytecode that implements such safety behaviors is compiled into native code, these behaviors must be reproduced as well, leading to an explosion of the code size. Indeed, a large part of the Java bytecode implement these safety mechanisms.

Although the runtime checks are necessary to the safety of the Java virtual machine, they are most of the time used as a protection mechanism against programming errors or malicious code: A runtime exception should be the result of an exceptional, unexpected program behavior and is rarely thrown when executing sane code - doing so is considered poor programming practice. The safety checks are therefore without effect most of the time, and, in the case of native code, uselessly bloat the code.

Several studies proposed to factorize these checks or in some case to eliminate them, but none proposed a complete elimination without hazarding the system security. In this paper, we use formal proofs to ensure that run-time checks can never be true into a program, which allows us to completely and safely eliminate them from the generated native code. The programs to optimize are JML-annotated against runtime exceptions and verified by the Java Applet Correctness Kit (JACK [4]). We have been able to remove almost all of the runtime checks on tested programs, and obtained native ARM thumb code which size was comparable to the original bytecode.

The remainder of this paper is organized as follows. In section 2, we overview the methods used for compiling Java bytecode into native code, and evaluate the previous work aiming at optimizing runtime exceptions in the native code. Then, section 3 describes our method for removing runtime exceptions on the basis of formal proofs. We experimentally evaluate this method in section 4, discuss its limitations in 5 and conclude in 6.

2 Java and Ahead-of-Time Compilation

Compiling Java into native code is a common practice in the embedded domain. This section gives an overview of the different compilation techniques of Java programs, and points out the issue of runtime exceptions. We are then looking at how existing solutions address this issue.

2.1 Ahead-of-Time & Just-in-Time Compilation

Ahead-of-Time (AOT) compilation is a common way to improve the efficiency of Java programs. It is related to Just-in-Time (JIT) compilation by the fact

that both processes take Java bytecode as input and produce native code that the architecture running the virtual machine can directly execute. AOT and JIT compilation differ by the time at which the compilation occurs. JIT compilation is done, as its name states, just-in-time by the virtual machine, and must therefore be performed within a short period of time which leaves little room for optimizations. The output of JIT compilation is machine-language. On the contrary, AOT compilation compiles the Java bytecode way before the program is run, and links the native code with the virtual machine. In other words, it translates non-native methods into native methods (usually C code) prior to the whole system execution. AOT compilers either compile the Java program entirely, resulting in a 100% native program without a Java interpreter, or can just compile a few important methods. In the latter case, the native code is usually linked with the virtual machine. AOT compilation has no or few time constraints, and can generate optimized code. Moreover, the generated code can take advantage of the C compiler's own optimizations.

JIT compilation in interesting by several points. For instance, there is no prior choice about which methods must be compiled: the virtual machine compiles a method when it appears that doing so is beneficial, e.g. because the method is called often. However, JIT compilation requires embedding a compiler within the virtual machine, which needs resources to work and writable memory to store the compiled methods. Moreover, the compiled methods are present twice in memory: once in bytecode form, and another time in compiled form. While this scheme is efficient for decently-powerful embedded devices such as PDAs, it is inapplicable to very restrained devices like smartcards or sensors. For them, ahead-of-time compilation is usually preferred because it does not require a particular support from the embedded virtual machine outside of the ability to run native methods, and avoids method duplication. AOT compilation has some constraints, too: the compiled methods must be known in advance, and dynamically-loading new native methods is forbidden, or at least very unsafe.

Both JIT and AOT compilers must produce code that exactly mimics the behavior of the Java virtual machine. In particular, the safety checks performed on some bytecodes must also be performed in the generated code.

2.2 Java Runtime Exceptions

The JVM (Java Virtual Machine) [5] specifies a safe execution environment for Java programs. Contrary to native execution, which does not automatically control the safety of the program's operations, the Java virtual machine ensures that every instruction operates safely. The Java environment may throw predefined runtime exceptions at runtime, like the following ones:

`NullPointerException` This exception is thrown when the program tries to dereference a `null` pointer. Among the instructions that may throw this

exceptions are: `getfield`, `putfield`, `invokevirtual`, `invokespecial`, and the set of *type*`astore` instructions[4].

`ArrayIndexOutOfBoundsException` If an array is accessed out of its bounds, this exception is thrown to prevent the program from accessing an illegal memory location. According to the Java Virtual Machine specification, the instructions of the family *type*`astore` and *type*`aload` may throw such an exception.

`ArithmeticException` This exception is thrown when exceptional arithmetic conditions are met. Actually, there is only one such case that may occur during runtime, namely the division of an integer by zero, which may be done by `idiv`, `irem`, `ldiv` and `lrem`.

`NegativeArraySizeException` Thrown when trying to allocate an array of negative size. `newarray`, `anewarray` and `multianewarray` may throw this exception.

`ArrayStoreException` Thrown when an object is attempted to be stored into an array of incompatible type. This exception may be thrown by the `aastore` instruction.

`ClassCastException` Thrown when attempting to cast an object to an incompatible type. The `checkcast` instruction may throw this exception.

`IllegalMonitorStateException` Thrown when the current thread is not the owner of a released monitor, typically by `monitorexit`.

If the JVM detects that executing the next instruction would result in an inconsistency or an illegal memory access, it throws a runtime exception, that may be caught by the current method or by other methods on the current stack. If the exception is not caught, the virtual machine exits. This safe execution mode implies that many checks are made during runtime to detect potential inconsistencies. For instance, the `aastore` bytecode, which stores an object reference into an array, may throw three different exceptions: `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `ArrayStoreException`.

Of the 202 bytecodes defined by the Java virtual machine specification, we noticed that 43 require at least one runtime exception check before being executed. While these checks are implicitly performed by the bytecode interpreter in the case of interpreted code, they must explicitly be issued every time such a bytecode is compiled into native code, which leads to a code size explosion. Ishizaki et al. measured that bytecodes requiring runtime checks are frequent in Java programs: for instance, the natively-compiled version of the SPECjvm98 `compress` benchmark has 2964 exception check sites for a size of 23598 bytes. As for the `mpegaudio` benchmark, it weights 38204 bytes and includes 6838 exception sites [6]. The exception check sites therefore make a non-neglectable part of the compiled code.

Figure 1 shows an example of Java bytecode that requires a runtime check to be issued when being compiled into native code.

[4] the JVM instructions are parametrized, thus we denote by *type*`astore` the set of array store instructions, which includes `iastore`, `sastore`, `lastore`, ...

Java version:

```
iload i
iload j
idiv
ireturn
```

C version:

```
1   int i, j;
2   if (j == 0)
3     THROW(ArithmeticException);
4   RETURN_INT(i / j);
```

Fig. 1. A Java bytecode program and its (simplified) C-compiled version. The behavior of the division operator in Java must be entirely reproduced by the C program, which leads to the generation of a runtime exception check site

It is, however, possible to eliminate these checks from the native code if the execution context of the bytecode shows that the exceptional case never happens. In the program of figure 1, the lines 2 and 3 could have been omitted if we were sure that for all possible program paths, `j` can never be equal to zero at this point. This allows to generate less code and thus to save memory. Removing exception check sites is a topic that has largely been studied in the domain of JIT and AOT compilation.

2.3 Related Work

Toba [7] is a Java-to-C compiler that transforms a whole Java program into a native one. Harissa [8] is a Java environment that includes a Java-to-C compiler as well as a virtual machine, and therefore supports mixed execution. While both environments implement some optimizations, they are not able to detect and remove unused runtime checks during ahead-of-time compilation. The "Java? C!" (JC[5]) Virtual Machine [9] is a Java virtual machine implementation that converts class files into C code using the Soot [10] framework, and runs their compiled version. It supports redundant exceptions checks removal, and is tuned for runtime performance, by using operating system signals in order to detect exceptional conditions like null pointer dereferencing. This allows to automatically remove most of the `NullPointerException`-related checks.

In [11] and [12], Hummel et al. use a Java compiler that annotates bytecodes with higher-level information known during compile-time in order to improve the efficiency of generated native code. [6] proposes methods for optimizing exceptions handling in the case of JIT compiled native code. These works rely on knowledge that can be statically inferred either by the Java compiler or by the JIT compiler. In doing so, they manage to efficiently factorize runtime checks, or in some cases to remove them. However, they are still limited to the context of the compiled method, and do not take the whole program into account. Indeed, knowing properties about a the parameters of a method can help removing further checks.

[5] In the remainder of this paper, the *JC* abbreviation is always used to refer to the *"Java? C!"* virtual machine, and never to *JavaCard*

We propose to go further than these approaches, by giving more precise directives as to how the program behaves in the form of JML annotations. These annotations are then used to get formal behavioral proofs of the program, which guarantee that runtime checks can safely be eliminated for ahead-of-time compilation.

3 Optimizing Ahead-of-Time Compiled Java Code

For verifying the bytecode that will be compiled into native code, we use the JACK verification framework (short for Java Applet Correctness Kit). JACK is designed as a plugin for the Eclipse interface development environment. It supports both the Java Modeling Language (JML [13]) and the ByteCode Specification Language (BCSL [14]), respectively at source and bytecode level, and also supplies a compiler from JML to BCSL. The tool supports only the sequential subset of the Java and Java bytecode languages, but this is sufficient for the purpose of the present paper. Thus, from a Java program annotated with JML or a bytecode program annotated with BCSL, JACK generates proof obligations at the source or bytecode level respectively. JACK can then translate the resulting verification conditions for several theorem provers: Coq, Simplify, Atelier B.

Verifying that a bytecode program does not throw Runtime exceptions using JACK involves several stages:

1. Writing the JML specification at the source level of the application, which expresses that no runtime exceptions are thrown.
2. Compiling the Java sources and their JML specification[6].
3. Generating the verification conditions over the bytecode and its BCSL specification, and proving the verification conditions. During the calculation process of the verification conditions, they are indexed with the index of the instruction in the bytecode array they refer to and the type of specification they prove (e.g. that the proof obligation refers to the exceptional postcondition in case an exception of type `Exc` is thrown when executing the instruction at index `i` in the array of bytecode instructions of a given method). Once the verifications are proved, information about which instructions can be compiled without runtime checks is inserted in user defined attributes of the class file.
4. Using these class file attributes in order to optimize the generated native code. When a bytecode that has one or more runtime checks in its semantics is being compiled, the bytecode attribute is queried in order to make sure that the checks are necessary. If it indicates that the exceptional condition has been proved to never happen, then the runtime check is not generated.

Our approach benefits from the accurateness of the JML specification and from the bytecode verification condition generator. Performing the verification

[6] the BCSL specification is inserted in user defined attributes in the class file and so does not violate the class file format

over the bytecode allows to easily establish a relationship between the proof obligations generated over the bytecode and the bytecode instructions to optimize.

In the rest of this section, we explain in detail all the stages of the optimization procedure.

3.1 JML Annotations

JML is a rich behavioral interface specification language, similar to Java and designed for it, that follows the design by contract paradigm [15]. Among the features that JML supports and which we use in this study are:

Method preconditions The method precondition states what must hold when the method is called, i.e. the precondition must hold at every method call site.

Method postconditions JML allows to specify both the exceptional and normal terminations of a method. One can express which property should hold if a method terminates normally and which property should hold if a method terminates by throwing an exception. The exceptional and normal postconditions state what the method guarantees after its execution and are verified when establishing the correctness of the method implementation.

Class invariants These properties must be established at every visible program state. In particular, the property must hold before and after every method call. The class invariant is not required to hold before calling the class constructor, but must hold once the constructor returns.

Loop invariants and loop frame conditions A loop invariant is a predicate that must hold every time the corresponding loop entry is reached. The loop frame condition states which locations are modified by the loop.

3.2 Methodology for Writing A Specification Against Runtime Exceptions

We now illustrate with an example which annotations must be generated in order to check if a method may throw an exception. Figure 2[7] shows a Java method annotated with a JML specification. The method `clear` declared in class `Code_Table` receives an integer parameter `size` and assigns `0` to all the elements in the array field `tab` whose indexes are smaller than the value of the parameter `size`. The specification of the method guarantees that if every caller respects the method precondition and if every execution of the method guarantees its postcondition then the method `clear` never throws an exception of type or subtype `java.lang.Exception`[8]. This is expressed by the class and method specification contracts. First, a class invariant is declared which states that once an instance of type `Code_Table` is created, its array field `tab` is not null. The class invariant guarantees that no method will throw a `NullPointerException` when dereferencing (directly or indirectly) `tab`.

[7] although the analysis that we describe is on bytecode level, for the sake of readability, the examples are also given on source level

[8] Note that every Java runtime exception is a subclass of `java.lang.Exception`

```
final class Code_Table {
  private/*@spec_public */short tab[];

  //@invariant tab != null;

  ...

  //@requires size <= tab.length;
  //@ensures true;
  //@exsures (Exception) false;
  public void clear(int size) {
  1  int code;
  2  //@loop_modifies code, tab[*];
  3  //@loop_invariant code <= size && code >= 0;
  4  for (code = 0; code < size; code++) {
  5    tab[code] = 0;
     }
  }
}
```

Fig. 2. A JML-annotated method

The method precondition requires the `size` parameter to be smaller than the length of `tab`. The normal postcondition, introduced by the keyword `ensures`, basically says that the method will always terminate normally, by declaring that the set of final states in case of normal termination includes all the possible final states, i.e. that the predicate `true` holds after the method's normal execution[9]. On the other hand, the exceptional postcondition for the exception `java.lang.Exception` says that the method will not throw any exception of type `java.lang.Exception` (which includes all runtime exceptions). This is done by declaring that the set of final states in the exceptional termination case is empty, i.e. the predicate `false` holds if an exception caused the termination of the method. The loop invariant says that the array accesses are between index `0` and index `size - 1` of the array `tab`, which guarantees that no loop iteration will cause an `ArrayIndexOutOfBoundsException` since the precondition requires that `size <= tab.length`.

3.3 Compiling JML annotations into BCSL specifications

Once the source code is completed by the JML specification, the Java source is compiled using a normal non-optimizing Java compiler that generates debug information like *LineNumberTable* and *LocalVariableTable*, needed for compiling the JML annotations. From the resulting class file and the specified source file,

[9] Actually, after terminating execution the method guarantees that the first `size` elements of the array tab will be equal to 0, but as this information is not relevant to proving that the method will not throw runtime exceptions we omit it

the JML annotations are compiled into BCSL and inserted into user-defined attributes of the class file. Figure 3 gives the bytecode version of the `clear` method shown earlier and its BSCL specification. In the example, `lv[0]` stands for the `this` instance and `lv[1]` stands for the first parameter that the method receives. A detailed description of the JML compiler can be found in [14].

```
  //@invariant tab(lv[0]) != null;

  ...

  //@requires lv[1] <= length(tab(lv[0]));
  //@ensures true;
  //@exsures (Exception) false;

method clear

 0 iconst_0
 1 istore_2
 2 goto 15
 5 aload_0
 6 getfield tab
 9 iload_2
10 iconst_0
11 sastore
12 iinc 2 by 1
15 iload_2
16 iload_1
17 if_icmplt 5
20 return
```

Fig. 3. The specified bytecode of method `clear`

3.4 Generation of the Verification Conditions

In order to generate the verification conditions, we use a bytecode verification condition generator (vcGen) based on a bytecode weakest precondition calculus [14]. The weakest precondition function wp returns, for every instruction `ins`, normal postcondition ψ, and exceptional function ψ^{exc} the weakest predicate $wp(\texttt{ins}, \psi, \psi^{exc})$ such that if it holds in the pre-state of the instruction `ins` and if the instruction terminates normally, then the normal postcondition ψ holds in the poststate and if `ins` terminates on an exception `Exc`, then the predicate $\psi^{exc}(\texttt{Exc})$ holds. From the annotated bytecode the vcGen calculates a set of verification conditions for every method of the application. The verification conditions for a method are generated by tracing all the execution paths

in it starting at every `return`, `athrow` and loop end instruction up to reaching the method entry point. During the process of generation of the verification conditions, for every instruction that may throw a runtime exception a new verification condition is generated.

In figure 4, we show the weakest precondition rule for the `getfield` instruction. As the virtual machine is stack-based, the rule mentions the stack `stack` and the stack counter `cntr`, thus the stack top element is referred as `stack(cntr)`. If the top stack element `stack(cntr)` is not null, `getfield` pops `stack(cntr)` which is an object reference and pushes the value of the referenced field onto the operand stack in `stack(cntr)`. If the stack top element is null, the Java Virtual Machine specification says that the `getfield` instruction throws a `NullPointerException`.

When the verification condition generator works over a method, it labels the formula related to the exceptional termination of every instruction with the index of the instruction in the bytecode array of the method. For example, if a `getField` instruction is met in the bytecode of a method, a conjunction is generated and the conjunct related to the exception is labeled as shown by figure 4. Finally, indexing the verification conditions allows to identify later in the proof phase which instructions can be optimized.

Another important point is that the underlying vcGen is proved to be correct [14], thus our methodology also correctly performs optimizations.

$$
\begin{array}{l}
wp(ind : \texttt{getfield Cl.f},\ \psi,\ \psi^{exc}) = \\
\quad \texttt{stack(cntr)} \neq \texttt{null} \Rightarrow \\
\qquad \psi\,[\texttt{stack(cntr)} \leftarrow \texttt{Cl.f(stack(cntr))}] \\
\left(\begin{array}{l} \wedge \\ ind : \texttt{stack(cntr)} = \texttt{null} \Rightarrow \end{array} \right. \\
\qquad \left. \psi^{exc}(\texttt{NullPointerException}) \begin{array}{l} [\texttt{cntr} \leftarrow 0] \\ [\texttt{stack(0)} \leftarrow \texttt{ref}_{\texttt{NullPointer}}] \end{array} \right)
\end{array}
$$

Fig. 4. The weakest precondition rule for the `putfield` instruction

3.5 From Program Proofs to Program Optimizations

In this phase, the bytecode instructions that can safely be executed without runtime checks are identified. Depending on the complexity of the verification conditions, Jack can discharge them to the fully automatic prover Simplify, or to the Coq and AtelierB interactive theorem prover assistants.

There are several conditions to be met for a bytecode instruction to be optimized safely – the precondition of the method the instruction belongs to must hold every time the method is invoked, and the verification condition related to the exceptional termination must also hold. In order to give a flavor of the

verification conditions we deal with, figure 5 shows part of the verification condition related to the possible `ArrayIndexOutOfBounds` exceptional termination of instruction `11 sastore` in figure 3, which is actually provable.

$$
\begin{array}{l}
\ldots \\
\texttt{length}(\texttt{tab}(\texttt{lv[0]}) \leq \texttt{lv[2]}_{15} \vee \texttt{lv[2]}_{15} < 0 \\
\wedge \\
\texttt{lv[2]}_{15} \geq 0 \\
\wedge \\
\texttt{lv[2]}_{15} < \texttt{lv[1]} \\
\wedge \\
\texttt{lv[1]} \leq \texttt{length}(\texttt{tab}(\texttt{lv[0]}))
\end{array}
\quad \Rightarrow \mathit{false}
$$

Fig. 5. The verification condition for the `ArrayIndexOutOfBoundException` check related to the `sastore` instruction of figure 3

Once identified, proved instructions can be marked in user-defined attributes of the class file so that the compiler can find them.

3.6 More Precise Optimizations

As we discussed earlier, in order to optimize an instruction in a method body, the method precondition must be established at every call site and the method implementation must be proved not to throw an exception under the assumption that the method precondition holds. This means that if there is one call site where the method precondition is broken then no instruction in the method body will be optimized.

Actually, the analysis may be less conservative and therefore more precise. We illustrate with an example how one can achieve more precise results.

Consider the example of figure 6. On the left side of the figure, we show source code for method `setTo0` which sets the `buff` array element at index `k` to 0. On the right side, we show the bytecode of the same method. The `iastore` instruction at index `3` may throw two different runtime exceptions: `NullPointerException`, or `ArrayIndexOutOfBoundException`. For the method execution to be safe (i.e. no runtime exception is thrown), the method requires some conditions to be fulfilled by its callers. Thus, the method's precondition states that the `buff` array parameter must not be null and that the `k` parameter must be inside the bounds of `buff`. If at all call sites we can establish that the `buff` parameter is always different from null, but there are sites at which an unsafe parameter `k` is passed, the optimization for `NullPointerException` is still safe although the optimization for `ArrayIndexOutOfBoundException` is not possible. In order to obtain this kind of preciseness, a solution is to classify the preconditions of a method with respect to what kind of runtime exception they protect the code from. For our example, this classification consists of two groups of preconditions.

The first is related to `NullPointerException`, i.e. `buff != null` and the second consists of preconditions related to `ArrayIndexOutOfBoundException`, i.e. `k >= 0 && k <= buff.length`. Thus, if the preconditions of one group are established at all call sites, the optimizations concerning the respective exception can be performed even if the preconditions concerning other exceptions are not satisfied.

```
...

//@requires buff != null;
//@requires k >= 0 ;                        0 aload_2
//@requires k <= buff.length;               1 iload_1
//@ensures true;                            2 iconst_0
//@exsures (Exception) false;               3 iastore
public void setTo0(int k,int[] buff)        4 return
{
  buff[k] = 0;
}
```

Fig. 6. The source code and bytecode of a method that may throw several exceptions

4 Experimental Results

This section presents an application and evaluation of our method on various Java programs.

4.1 Methodology

We have measured the efficiency of our method on two kinds of programs, that implement features commonly met in restrained and embedded devices. `crypt` and `banking` are two smartcard-range applications. `crypt` is a cryptography benchmark from the Java Grande benchmarks suite, and `banking` is a little banking application with full JML annotations used in [4]. `scheduler` and `tcpip` are two embeddable system components written in Java, which are actually used in the JITS [16] platform. `scheduler` implements a threads scheduling mechanism, where scheduling policies are Java classes. `tcpip` is a TCP/IP stack entirely written in Java, that implements the TCP, UDP, IP, SLIP and ICMP protocols. These two components are written with low-footprint in mind ; however, the overall system performance would greatly benefit from having them available in native form, provided the memory footprint cost is not too important.

For every program, we have followed the methodology described in section 3 in order to prove that runtime exceptions are not thrown in these programs. We look at both the number of runtime exception check sites that we are able to remove from the native code, and the impact on the memory footprint of the

natively-compiled methods with respect to the unoptimized native version and the original bytecode. The memory footprint measurements were obtained by compiling the C source file generated by the JITS ahead-of-time (AOT) compiler using GCC 4.0.0 with optimization option `-Os`, for the ARM platform in thumb mode. The native methods sizes are obtained by inspecting the .o file with `nm`, and getting the size for the symbol corresponding to the native method.

Regarding the number of eliminated exception check sites, we also compare our results with the ones obtained using the JCk virtual machine mentioned in 2.3, version 1.4.6. The results were obtained by running the `jcgen` program on the benchmark classes, and counting the number of explicit exception check sites in the generated C code. We are not comparing the memory footprints obtained with the JITS and JC AOT compilers, for this result would not be relevant. Indeed, JC and JITS have very different ways to generate native code. JITS targets low memory footprint, and JC runtime performance. As a consequence, a runtime exception check site in JC is heavier than one in JITS, which would falsify the experiments. Suffices to say that our approach could be applied on any AOT compiler, and that the most relevant measurement is the number of runtime exception check sites that remains in the final binary - our measurements on the native code memory footprint are just here to evaluate the size impact of exception check sites.

4.2 Results

Table 1 shows the results obtained on the four tested programs. The three first columns indicate the number of check sites present in the bytecode, the number of explicit check sites emitted by JC, and the number of check sites that we were unable to prove useless and that must be present in our optimized AOT code. The last columns give the memory footprints of the bytecode, unoptimized native code, and native code from which all proved exception check sites are removed.

Table 1. Number of exception check sites and memory footprints when compiled for ARM thumb

Program	# of exception check sites			Memory footprint (bytes)		
	Bytecode	JC	Proven AOT	Bytecode	Naive AOT	Proven AOT
`crypt`	190	79	1	1256	5330	1592
`banking`	170	12	0	2320	5634	3582
`scheduler`	215	25	0	2208	5416	2504
`tcpip`	1893	288	0	15497	41540	18064

On all the tested programs, we were able to prove that all but one exception check site could be removed. The only site that we were unable to prove from `crypt` is linked to a division, which divisor is a computed value that we were unable to prove not equal to zero. JC has to retain 16% of all the exception

check sites, with a particular mention for `crypt`, which is mainly made of array accessed and has more remaining check sites.

The memory footprints obtained clearly show the heavy overhead induced by exception check sites. Despite of the fact that the exception throwing convention has deliberately been simplified for our experiments, optimized native code is less than half the size of the non-optimized native code. The native code of `crypt`, which heavily uses arrays, is actually made of exception checking code at 70%.

Comparing the size of the optimized native versions with the bytecode reveals that proved native code is just slightly bigger than bytecode. The native code of `crypt` is 27% bigger than its bytecode version. Native `scheduler` only weights 13.5% more that its bytecode, `tcpip` 16.5%, while `banking` is 54% heavier. This last result is explained by the fact that, being an application and not a system componant, `banking` includes many native-to-java method invocations for calling system services. The native-to-java calling convention is costly in JITS, which artificially increases the result.

Finally, table 2 details the human work required to obtain the proofs on the benchmark programs, by comparing the amount of JML code with respect to the comments-free source code of the programs. It also details how many lemmas had to be manually proved.

Table 2. Human work on the tested programs

Program	Source code size (bytes)		Proved lemmas	
	Code	JML	Automatically	Manually
`crypt`	4113	1882	227	77
`banking`	11845	15775	379	159
`scheduler`	12539	3399	226	49
`tcpip`	83017	15379	2233	2191

On the three programs that are annotated for the unique purpose of our study, the JML overhead is about 30% of the code size. The `banking` program was annotated in order to prove other properties, and because of this is made of more JML annotations than actual code. Most of the lemmas could be proved by Simplify, but a non-neglectable part needed human-assistance with Coq. The most demanding application was the TCP/IP stack. Because of its complexity, nearly half of the lemmas could not be proved automatically.

The gain in terms of memory footprint obtained using our approach is therefore real. One may also wonder whether the runtime performance of such optimized methods would be increased. We did the measurements, and only noticed a very slight, almost undetectable, improvement of the execution speed of the programs. This is explained by the fact that the exception check sites conditions are always false when evaluated, and therefore the amount of supplementary code executed is very low. The bodies of the proved runtime exception check sites are, actually, dead code that is never executed.

5 Limitations

Our approach suffers from some limitations and usage restrictions, regarding its application on multi-threaded programs and in combination with dynamic code loading.

5.1 Multi-Threaded Programs

As we said in section 3, JACK only supports the sequential subset of Java. Because of this, we are unable to prove check sites related to monitor state checking, that typically throws an `IllegalMonitorStateException`. However, they can be simplified if it is known that the system will never run more than one thread simultaneously. It should be noted, that Java Card does not make use of multi-threading and thus doesn't suffer from this limitation.

5.2 Dynamic Code Loading

Our removal of runtime exception check sites is based on the assumption that a method's preconditions are always respected at all its call sites. For closed systems, it is easy to verify this property, but in the case of open systems which may load and execute any kind of code, the property could not always be ensured. In the case where the set of applications that will run on the system is not statically known, our approach could not be safely applied on public methods since dynamically-loaded code may call them without respecting their preconditions.

5.3 Implications Regarding Security

In addition to the two limitations mentioned above, one should also be aware that our method doesn't protect the system from errors injections in the code through hardware attacks. Suppressing dynamic checking on systems that are subject to such attacks would potentially open a security breach.

6 Conclusion

The main contribution of the present article is a new Java-to-native code optimization technique based on static program verification using formal methods. The methodology gives more precise and therefore better results than other existing solutions in the field and allows us to remove almost all the exception check sites in the native code, as we show in section 4. The memory footprints of natively-compiled methods thus become comparable with the ones of the original bytecode when compiled in ARM thumb.

Although we applied this work to the ahead-of-time compilation of Java methods, the bytecode annotations could also be interpreted by JIT compilers, which would then also be able to completely get rid of a considerable part of runtime exceptions.

Acknowledgments

The authors would like to thank Jean-Louis Lanet for kindly providing us with the JML-annotated sources of the `banking`, `scheduler` and `tcpip` programs evaluated in this paper.

References

1. D. Mulchandani, "Java for embedded systems," *Internet Computing, IEEE*, vol. 2, no. 3, pp. 30 – 39, 1998.
2. L. Lagosanto, "Next-generation embedded java operating system for smart cards," in *4th Gemplus Developer Conference*, 2002.
3. G. Grimaud and J.-J. Vandewalle, "Introducing research issues for next generation Java-based smart card platforms," in *Proc. Smart Objects Conference (sOc'2003)*, (Grenoble, France), 2003.
4. L. Burdy, A. Requet, and J.-L. Lanet, "Java applet correctness: A developer-oriented approach," in *FME 2003: Formal Methods: International Symposium of Formal Methods Europe* (K. Araki, S. Gnesi, and D. Mandrioli, eds.), vol. 2805, pp. 422–439, 2003.
5. T. Lindholm and F. Yellin, *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.
6. K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani, "Design, implementation, and evaluation of optimizations in a just-in-time compiler," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, (New York, NY, USA), pp. 119–128, ACM Press, 1999.
7. T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for applications: A way ahead of time (wat) compiler," in *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, (Portland, Oregon), University of Arizona, June 1997.
8. G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: a flexible and efficient java environment mixing bytecode and compiled code," in *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon: USENIX, June 1997.
9. "JC Virtual Machine." http://jcvm.sourceforge.net/.
10. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - a java optimization framework," in *Proceedings of CASCON 1999*, pp. 125–135, 1999.
11. J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau, "Annotating the Java bytecodes in support of optimization," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1003–1016, 1997.
12. A. Azevedo, A. Nicolau, and J. Hummel, "Java annotation-aware just-in-time (ajit) complilation system," in *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, (New York, NY, USA), pp. 142–151, ACM Press, 1999.
13. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Mller, and J. Kiniry, *JML Reference Manual*, July 2005.
14. M. Pavlova, "Java bytecode logic and specification," tech. rep., INRIA, Sophia-Antipolis, 2005. Draft version.
15. B.Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2 revised ed., 1997.
16. "Java In The Small." http://www.lifl.fr/RD2P/JITS/.