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

Alexandre Courbot, Mariela Pavlova, Gilles Grimaud, Jean-Jacques Vandewalle

Seventh Smart Card Research and Advanced Application IFIP Conference, April 19th, 2006

Outline

Java-to-Native Compilation

Compiling Java bytecode into native code brings runtime advantages:

- Faster execution
- Especially beneficial for restrained systems with non-sophisticated JVMs

But Java-to-Native compilation also comes with drawbacks:

- Native code is typically 3 to 4 times bigger than bytecode,

Why is Native Code so Huge? An Example

The *idiv* bytecode throws an `ArithmeticException` if the divisor is equal to zero:

```
1 iload i
2 iload j
3 idiv
4 ireturn
```

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

There are many checks of this kind:

- Checking a pointer is not *null* before dereferencing it,
- Checking an array is accessed inside its bounds,
- Checking an array is created with a positive size,
- Checking affected types are compatible,
- ...

SPECjvm98: 2964 exception check sites for a native size of 23598 bytes (Ishizaki et al.).

Why is Native Code so Huge? An Example

The *idiv* bytecode throws an `ArithmeticException` if the divisor is equal to zero:

```
1 iload i
2 iload j
3 idiv
4 ireturn
```

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

There are many checks of this kind:

- Checking a pointer is not *null* before dereferencing it,
- Checking an array is accessed inside its bounds,
- Checking an array is created with a positive size,
- Checking affected types are compatible,
- ...

SPECjvm98: 2964 exception check sites for a native size of 23598 bytes (Ishizaki et al.).

Compilation Times

Java-to-Native compilation is typically done at two moments:

Ahead-Of-Time:

- Performed off-board,
- Methods to compile must be selected in advance,
- No or little time constraints,
- Native code replaces the bytecode,

Just-In-Time:

- Performed by an on-board compiler,
- Methods to compile are chosen by the JVM,
- Little time to perform optimizations,
- Native code must be stored in writable memory in addition to the bytecode

In either cases, the runtime exceptions check sites must be issued.

Suppressing Exceptions Check Sites

Runtime exceptions are (usually) a safety against programming errors. They should not be triggered by sane code.
We propose to formally prove that runtime exceptions are never thrown by a program.
Methodology:

1. Annotate source code with JML specification to express that no runtime exception will be thrown
2. Compile JML specification into BCSL as user-defined class file attributes
3. Generate and prove verification conditions over the bytecode and BCSL
4. Annotate class files with useless runtime exception check sites attribute

Annotating Source with JML

Write annotations that express no runtime exception is thrown.

```
1  private /*@spec_public */ short tab[];
2  //@invariant tab != null;
3  //@requires size <= tab.length;
4  //@ensures true;
5  //@exsures (Exception) false;
6  public void clear(int size) {
7    int code;
8    //@loop_modifies code, tab[*];
9    //@loop_invariant code <= size && code >= 0;
10   for (code = 0; code < size; code++) {
11     tab[code] = 0;
12   }
13 }
```

Lines 2 and 3 specifies what the method requires from its callers

Annotating Source with JML

Write annotations that express no runtime exception is thrown.

```
1  private/*@spec_public */short tab[];
2  //@invariant tab != null;
3  //@requires size <= tab.length;
4  //@ensures true;
5  //@exsures (Exception) false;
6  public void clear(int size) {
7    int code;
8    //@loop_modifies code, tab[*];
9    //@loop_invariant code <= size && code >= 0;
10   for (code = 0; code < size; code++) {
11     tab[code] = 0;
12   }
13 }
```

Lines 4 and 5 specifies what the method guarantees to its callers

Annotating Source with JML

Write annotations that express no runtime exception is thrown.

```
1  private /*@spec_public */ short tab[];
2  //@invariant tab != null;
3  //@requires size <= tab.length;
4  //@ensures true;
5  //@exsures (Exception) false;
6  public void clear(int size) {
7    int code;
8    //@loop_modifies code, tab[*];
9    //@loop_invariant code <= size && code >= 0;
10   for (code = 0; code < size; code++) {
11     tab[code] = 0;
12   }
13 }
```

Lines 8 and 9 specifies the loop invariants

Compile JML annotations into BCSL attributes

Class files are enriched with a user-defined BCSL attribute that expresses the JML specification.

```
1  //@invariant tab(lv[0]) != null;
2  //@requires lv[1] <= length(tab(lv[0]));
3  //@ensures true;
4  //@exsures (Exception) false;
5
6  method clear
7    iconst_0
8    istore_2
9    ...
10   return
```

Generating the Verification Conditions

Annotations are verified by the Java Applet Correction Kit (JACK).
Verification conditions are generated using a weakest precondition calculus.
They are then verified using a theorem prover (Simplify, Coq, AtelierB, ...)
A method can be optimized if:

- The precondition is respected at every call site,
- The normal and exceptional postconditions always hold.

Optimizing the Native Code

When a runtime exception throwing bytecode is met by the compiler, the useless exception sites attribute is checked with the bytecode index to see if the exception site is needed.

- Cheap and efficient for both ahead-of-time and just-in-time compilations
- The annotations must be trusted!

Experimental Results

Number of necessary exception check sites

Program	# of exception check sites		
	Bytecode	JC	Proven AOT
`crypt`	190	79	1
`banking`	170	12	0
`scheduler`	215	25	0
`tcpip`	1893	288	0

Program	Memory footprint (bytes)		
	Bytecode	Naive AOT	Proven AOT
`crypt`	1256	5330	1592
`banking`	2320	5634	3582
`scheduler`	2208	5416	2504
`tcpip`	15497	41540	18064

Experimental Results

Human work

Program	Source code size (bytes)	
	Code	JML
`crypt`	4113	1882
`banking`	11845	15775
`scheduler`	12539	3399
`tcpip`	83017	15379

Program	Proved lemmas	
	Automatically	Manually
`crypt`	227	77
`banking`	379	159
`scheduler`	226	49
`tcpip`	2233	2191

Conclusion

We presented a way to dramatically reduce the size of native code generated through Java bytecode by removing runtime exception check sites.

- JML annotations are used to generate verification conditions over runtime exceptions
- Sites are removed only if it is formally proved they are not useful
- Final size nearly as compact as original bytecode for ARM thumb!

Limitations:

- Right now, we are unable to remove exception check sites related to monitors
- Dynamic class loading is not safe if the loaded classes call optimized native code
- Human work still rather consequent, requires a verification expert to make it through!