Evaluation of a Java Ahead-of-Time Compiler for Embedded Systems†

DONG-HEON JUNG, SOO-MOOK MOON∗ AND SUNG-HWAN BAE

School of Electrical Engineering and Computer Science, Seoul National University, Seoul 151-742, Korea
∗Corresponding author: smoon@snu.ac.kr

Java embedded systems often include Java middleware classes installed on the client device. For higher performance, they can be compiled into machine code before runtime using an ahead-of-time compiler (AOTC). There are many approaches to AOTC; yet a bytecode-to-C (b-to-C) AOTC which translates the bytecode into the C code and then compiles it using an existing optimizing compiler such as gcc would be the most straightforward one. This paper explores a few important design and optimization issues of a b-to-C AOTC, including the compilation form for the translated C code, the call interfaces among translated and interpreted Java methods, and Java-specific optimizations by the AOTC that can complement the gcc optimizations. We evaluate these issues with our b-to-C AOTC implemented on the MIPS platform for the Sun’s CDC VM to understand their performance impact.

Keywords: Java ahead-of-time compiler; bytecode-to-C; embedded systems; Java virtual machine; code optimizations

Received 2 October 2010; revised 12 June 2011
Handling editor: Manfred Broy

1. INTRODUCTION

Many embedded systems including mobile phones, digital TVs (DTVs) and blue-ray disks employ Java as a standard software platform, due to its support for platform independence, security and faster development of reliable software contents. Platform independence is achieved by installing the Java virtual machine (JVM) on each platform [1], which executes Java’s compiled executable called bytecode via interpretation [2]. Since this software-based execution is much slower than hardware-based execution, compilation techniques that translate the bytecode into machine code have been employed, such as the just-in-time compiler (JITC) [3, 4] or the ahead-of-time compiler (AOTC) [5–20]. JITC and AOTC perform the translation during runtime and before runtime, respectively. AOTC is more advantageous in embedded systems since it obviates the runtime translation and memory overhead of JITC, which would waste their limited computing power and memory. Moreover, AOTC can optimize the code better than JITC due to its off-line optimizations, while online optimization in JITC cannot employ aggressive, time-consuming techniques. On the other hand, AOTC is not applicable to dynamically loaded classes such as xlets in DTVs [21] or midlets in mobile phones [22].

The best example of embedded Java classes where AOTC is applicable is the Java middleware (and system classes) installed on the client devices. For example the software platform for DTVs includes a Java middleware called OCAP (Open Cable Application Platform) or ACAP (Advanced Common Application Platform), which is installed on the DTV set-top box in advance [21]. Also, mobile phones include the MIDP middleware [22] on the handset. Blu-ray disks have the BD-J middleware [23] on the BD player. It will be more useful to compile these classes ahead-of-time with maximal optimizations, rather than resorting to JITC.

The Java middleware is getting more substantial to provide more functionality. In mobile phones, for example, the first MIDP middleware provided libraries for user interfaces only, yet its successor middleware called JTWI [24] provided an integrated library with music players and SMS. Now a more substantial middleware called MSA [25] is being introduced with more features. This increased functionality reduces the size and the complexity of downloaded classes as well. In fact, if the middleware is designed well, most of the running time should be spent on the middleware rather than on downloaded classes. We confirmed this by studying a DTV environment in Korea.
using the data broadcasting of a channel [26]. There are four menus in its xlet application including stock, news, weather and traffic information. We analyze the number of methods called during the execution of each menu, i.e. those methods executed until the first picture appears on the TV screen when we select the menu. We found that more than 90% of methods executed belong to the system classes or the ACAP classes, and xlet methods take a small portion. Consequently, accelerating the middleware/system using the AOTC instead of using the JITC or the interpreter can be justified.

There are two major approaches to AOTC depending on where the AOTC is performed. One is performing AOTC on the client device using the JITC included in the JVM, which we call the client-AOTC. For example Sun’s PhoneMe Advanced (MR2 version) includes an AOT option that allows compiling a list of pre-chosen methods using the JITC and saves their machine code on a persistent memory such as flash memory [27]. When the JVM is officially invoked, it will first load the machine code into the code JVM where JITC methods are supposed to be saved. There is also a client-AOTC, where at the end of a regular, JITC-based execution, we save the machine code in the code cache into the flash memory such that when they are used in a later, different run, their machine code is loaded directly to the code cache on an as-needed basis without JITC [28]. The biggest problem of the client-AOTC is its performance, limited by JITC-based optimization. Since the JITC overhead is part of the total running time, JITC cannot employ powerful optimization techniques such as those used in static compilers. Therefore, the benefit of using client-AOTC would be somewhat limited, only to removing the JITC overhead (the time used for JITC itself) or maybe removing the interpretation time to detect hot methods as well, if adaptive JITC [29] is employed.

The other approach is performing AOTC in the server using an off-line compiler, and installing the machine code on the client device. We call it the server-AOTC. There are two ways to server-AOTC. One is translating the bytecode directly into the machine code (bytecode-to-native) [18–20], and the other is translating the bytecode into the C code, which is then compiled into the machine code [5–17] by an existing compiler such as gcc [30]. This bytecode-to-C (b-to-C) approach is more practical since it allows faster development by using the full optimizations of an existing compiler and its debugging and profiling tools. Also, it allows a portable AOTC that can be used for multiple platforms without porting. In fact, many real-time JVMs take this approach [8–17], including commercial JVMs such as Jamaica [12, 13], IBM WebSphere Real-time VM [14], PERC [15] and Fiji [16, 17].

There are a few important design and optimization issues in the b-to-C AOTC. We first need to decide if we compile and link all the translated C code together with the JVM source code into a single static executable, or if we compile each of them separately and have the JVM load them dynamically on an as-needed basis at runtime. The former will require rebuilding of a new executable if there is an update on the middleware, while the latter will suffer from inefficiencies when there are many calls and accesses across compilation units or to the JVM functions since they should be resolved and linked at runtime.

Another issue is deciding an efficient method call interface so as to translate method calls between Java methods efficiently, since the total call overhead takes a significant portion of the whole running time in Java. This is especially important when we have a hybrid environment of AOTC and interpreter (or JITC), where the middleware is handled by AOTC while downloaded classes are executed by interpretation (or JITC). Since the call overhead between AOTC methods and interpreted (or JITC) methods will be higher than that between the same types of methods [31], we need to define an efficient call interface.

Finally, existing gcc optimizations might not be enough to generate efficient machine code since the C compiler is not aware that the translated C code is actually from the Java bytecode. So, the AOTC needs to perform some Java-specific optimizations, such as the elimination of redundant null pointer checks or array bounds checks (ABCs), which can complement the gcc optimizations.

This paper addresses these issues and evaluates them using our b-to-C AOTC developed for Sun’s CDC VM (CVM) [32]. We discuss the design choices we made for the AOTC, compared with other alternatives. Since our AOTC takes a similar approach to others, our evaluation of many trade-offs in designing and implementing the b-to-C AOTC will be generally useful and applicable to others. As far as we know, there has been little previous work in evaluating such design and implementation choices of the b-to-C AOTC comprehensively as in this paper. The rest of the paper is organized as follows. Section 2 describes how our b-to-C AOTC translates the bytecode into C code. Section 3 explores the design alternatives of the AOTC. Section 4 shows our evaluation results. Related work is in Section 5 and the summary is in Section 6.

2. OVERVIEW OF THE JVM AND OUR B-TO-C TRANSLATION

The Java VM is a typed stack machine [1]. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between the local variables and the operand stack. Each thread of execution has its own Java stack where a new frame is pushed when a method is invoked and is popped when it returns. A frame includes its state information (e.g. frame type: AOTC frame or interpreter frame), local variables and the operand stack. Method parameters are also local variables which are initialized to the actual parameters by the JVM.

Our b-to-C AOTC translates the bytecode into the C code with Java-specific optimizations, which is then compiled using gcc. Our AOTC can either translate all methods or selectively translate methods using a profile feedback in order to reduce...
the executable size. This is fine since our AOTC can work with the interpreter (or the JITC), and so AOTC methods and interpreted methods can be mixed in execution in an interoperable manner. Our b-to-C AOTC first analyze the bytecode and decides the C variables that need to be declared. Each local variable and each stack slot is translated into a C variable (which is called a local C variable and a stack C variable, respectively). Since the same stack slot can be pushed with differently typed value during execution, a type name is attached into a stack C variable name such that a stack slot can be translated into multiple C variables. For example, S0_ref is a C variable corresponding to a reference-type stack slot 0, while S0_int is a C variable corresponding to an integer-type stack slot 0.

The AOTC then translates each bytecode one by one into a corresponding C statement, with the status of the operand stack including the stack top pointer being kept track of. For example, iload_1 which pushes an integer-type local variable 1 onto the stack top is translated into a C statement S0_int=L1_int if the current stack top pointer points to the slot 0 when this bytecode is translated; then, the current stack top pointer is changed to point to the slot 1. If the next bytecode were iload_2, it would be translated to the C statement S1_int=L2_int. and the current stack top pointer now points to the slot 2. Figure 1 shows an example of translation from a bytecode compiled from a Java method to a C function.

Our AOTC supports precise garbage collection (GC) by generating additional C statements that collect root information [33]. It also includes an efficient exception handling mechanism using exception checks rather than using setjmp/longjumps [34]. In fact, the first argument of every function is CVMEExcEnv *ee, as shown in the example of Fig. 1, which contains information on the interpreter stack and data structures for exception handling and GC, etc.

3. DESIGN AND OPTIMIZATION OF THE B-TO-C AOTC

In this section, we discuss some of the important design and optimization issues of the b-to-C AOTC and explain the choices we made in our implementation, compared with other alternatives.

3.1. Compilation of the translated C code

We first need to decide the way to compile the translated C code. We would naturally make one translated C file per one Java class file, instead of making one big C file that includes the translated code of all class files. Therefore, a C file will include calls and accesses to methods and data fields defined in other C files as well as those defined in the same C file. We should make those cross-file calls and accesses as efficient as intra-file ones.

Similarly, the translated C code will often include calls and accesses to functions and data structures defined in the JVM, which should also be efficient. For example we can call JVM’s functions to handle locks (monitorenter/monitorexit), to check class information (checkcast/instanceof) or to allocate objects/arrays (new/newarray). Also, each thread checks if there is any pending request for GC from other threads at each point where GC can possibly occur (one example is a loop backedge as shown in Fig. 1c), which requires accessing a JVM global variable. These accesses should also be done efficiently.

Another issue to consider is the update of the middleware, which would require the retranslation and the recompilation of the updated class files. So, if the compilation unit is too large, the update/add of classes will incur substantial recompilation.

There are two major compilation approaches: separate compilation and combined compilation, as depicted in Fig. 2.
3.1.1. Separate compilation
One approach is compiling each translated C file separately from other C files and the JVM as shown in Fig. 2a, so JVM executable and multiple compiled AOTC classes are produced as an output. The JVM will load compiled classes dynamically on an as-needed basis at runtime. This is efficient when there is an update of class files since we only need to recompile the affected files. This is also efficient when new class files are added since their translated and compiled files can readily work with other class files without any special handling. In fact, there is a commercial Java platform for mobile phones called WIPI (Wireless Mobile Platform for Interoperability) [35] which takes this approach of compilation. In WIPI, not only the Java middleware on the mobile phone but the downloaded classes from the service provider are compiled into binaries by the b-to-C AOTC, which can then be downloaded into mobile phones. Both the middleware classes and the downloaded classes are compiled into separate binary files, so newly downloaded classes or updated classes can readily work with the existing ones and the JVM without any special handling.

On the other hand, this compilation approach has two problems. One is that inlining across classes is impossible. The other is that we need runtime resolution and linking. Since each translated C file is compiled separately, if there is a call or an access to another translated C file or to JVM internals, its address cannot be determined at compilation time. Therefore, a dynamic resolution and linking is required to patch the undetermined addresses. To reduce the patch overhead, out-of-class calls and accesses can be handled indirectly using a central table as shown in Fig. 3a [36]. It includes an entry for each unique unresolved reference, which is resolved and replaced by a real address (for static method calls or static field accesses) or a real offset (for virtual methods and instance field accesses) at runtime. The machine code generated with an unresolved reference includes a load from the indirection table to get the address or the offset. Figure 3a shows an example translated machine code for getstatic.

3.1.2. Combined compilation
The other approach is compiling and linking all the translated C files together with the JVM source code into a single static executable, as shown in Fig. 2b. In this case, a static method call is translated directly into a C function call and a static field access is translated into a direct access to the global variable corresponding to the static field, so gcc can link them together irrespective of whether it is a cross-file one or intra-file one. For virtual method calls and instance field accesses, the offsets within the method table and within the object can be known at compile time by analyzing all C files by the AOTC, so that an efficient code sequence can be generated. The JVM function calls and data accesses can be resolved and linked by the gcc, so that they can be efficient as well. Consequently, we do not need any additional load from the indirection table as in separate compilation. Figure 3b shows an example of the translated machine code for getstatic. The AOTC can also perform inlining freely, even across classes.

On the other hand, if there is an update of a class file or an addition of a new class file, we need to rebuild a new executable, which would be somewhat cumbersome. However, an update for the middleware would not be a frequent event, while the downloaded classes are handled by the JITC or the interpreter, and so this approach is more suitable for the AOTC of the middleware.
3.2. Method call interface

Our AOTC translates a Java method into a C function and a Java method call into a C function call. The first issue for the translation is how to define the C function names. While Java allows overloading of the same method name as well as the overriding of the same method signature in the class hierarchy, C does not allow them. So, we must define a unique C function name for each Java method. We follow the naming convention of JNI (Java native interface) [1] such that we define a function name based on the class name and the method signature. For static method calls, this name is used directly in the C function call statement. For virtual method calls, the call is made indirectly using a function pointer saved in the method block.

A more important issue is how to pass arguments and the return value in the translated C code. In the original interpreter mode of execution, method calls are made by pushing arguments on the caller’s operand stack first, which then become the local variables of the callee. When the callee returns, the return value is pushed on the caller’s operand stack. The AOTC call interface does not have to mimic this calling convention strictly but should satisfy the expected call behavior among AOTC methods. Moreover, since there can be calls between AOTC methods and interpreted methods (e.g. an interpreted, downloaded class method can make a call to an AOTCed, middleware method), the call interface should accommodate such calls. There are four possible solutions.

3.2.1. Standard C call interface

The first solution is simply using the standard C function call interface. The C variables of the caller method corresponding to the arguments (which are variables representing the operand stack slots, such as $S_0$ _int_, $S_1$ _ref_, etc., since arguments should be on the operand stack) are directly used as actual parameters of the C function call. The C variables of the callee method corresponding to its local variables (such as $L_0$ _int_, $L_1$ _ref_, etc.) are used as formal parameters in the callee function declaration since arguments should become local variables. Similarly, the C variable of the callee holding the return value is used in the return statement, which will be saved in the C variable of the caller corresponding to the operand stack slot 0 (such as $S_0$ _int_). Figure 4a-1 and c-1 show a part of the translated C code for the caller and the callee methods, respectively, when the callee is a static method. For a static method call, it can be known at AOTC time which method will be called, and so it is translated into a direct C function call (callee_method() in Fig. 4a-1). For a virtual method call, we cannot know which method will be called at runtime or if the method is AOTCed or not. So we should generate a sequence of C statements, which read the method table to access the method block, check if the method is AOTCed and make a function call using the corresponding function pointer, as shown in Fig. 4a-2. The corresponding callee method is shown in Fig. 4c-2. It should be noted that a virtual method requires one more argument for this pointer, reserved as the first argument after ee (hereafter we shall explain the case of the static methods only).

This solution allows the fastest AOTC-to-AOTC calls, leading to the best performance for a full-AOTC environment where all executed methods are AOTCed. For a partial-AOTC environment where the downloaded classes are interpreted while the middleware classes are AOTCed, there is an overhead when an interpreted method (or a JITC method) calls an AOTC method. For these interpreter-to-AOTC calls, the interpreter must copy arguments from the Java operand stack to registers or the C stack of the AOTC methods using an assembly routine (it is actually implemented by a separate function), as shown in Fig. 4b. This is similar to the implementation of the interpreter-to-JNI calls, although our translated C code is not based on JNI. The argument copying requires a loop iterating through the argument list, testing their types and processing of the argument data based on their types (e.g. read two words from the operand stack if the argument is a 64-bit data) as well as the actual copying, which would cause a non-trivial overhead if there are many interpreter-to-AOTC calls in the partial-AOTC environment.
3.2.2. Operand stack call interface

To reduce the overhead of calls from interpreted methods to AOTC methods, we can simply follow the interpreter’s calling convention strictly such that the interpreted caller method passes a pointer of the operand stack to the AOTC callee method, as shown in Fig. 5b. The callee method retrieves arguments from the operand stack and copies them to the local C variables (we know the type and the number of arguments during the AOTC time), and there is a statement in the callee method copying the return value in the caller’s operand stack in Fig. 5c-1. We call this the operand stack interface, which is similarly implemented to the interpreter-to-CNI (compiled native interface) calls (CNI is a native interface used in CVM for allowing a faster native method implementation than JNI, which can work in a GC-unsafe mode while JNI works in a GC-safe mode).

It might be questioned if this retrieving overhead in the AOTC callee in Fig. 5c-1 is equivalent to the copying overhead in the interpreted caller in Fig. 4b for the standard C interface since we copy/retrieve the same number of arguments. However, the argument copying in Fig. 4b includes an additional overhead of loop iterations and argument testing which is not present in the argument retrieving in Fig. 5c-1. Moreover, the operand stack pointer is likely to be passed via a register no matter how many arguments there are, whereas only up to four arguments can be passed via registers in argument copying and additional arguments are pushed on the C stack in MIPS (in fact, one
argument is already reserved for the env variable and another one is for the this pointer in case of a virtual call, so real arguments are likely to be passed via the stack); this can reduce the argument passing overhead itself. Consequently, the interpreter-to-AOTC calls can be accelerated with the operand stack interface.

Unfortunately, the AOTC-to-AOTC calls are slower with this interface. The AOTC caller should explicitly copy argument C variables to the operand stack and pass its pointer to the callee method, as shown in Fig. 5a-1.

3.2.3. Mixed call interface

The standard C call interface is beneficial when an AOTC method calls an AOTC method while the operand stack call interface is advantageous when an interpreted method calls an AOTC method. We can devise a mixed call interface such that we use the former scheme when an AOTC method calls an AOTC method while we use the latter scheme when an interpreted method calls an AOTC method. To implement this mixed scheme, we follow the former scheme, but add a new argument that is a pointer to the operand stack. When this argument is NULL, it means that the caller is an AOTC method, requiring no special action; otherwise the caller is an interpreted method and we need to copy the arguments from the operand stack to the local C variables. The AOTC caller, the interpreted caller and the AOTC callee with this mixed scheme are shown in Fig. 6.

3.2.4. Wrapper function interface

Although the mixed call interface can exploit the benefits of both the operand stack interface and the standard C interface, the overhead of checking if the caller is AOTCed or interpreted is non-trivial, especially for the AOTC-to-AOTC calls compared with the standard C interface. Therefore, we propose an enhanced version of the standard C interface as follows. For each AOTC method, we generate a C function based on the standard C interface as previously. This original AOTC C function will be invoked by an AOTC caller, achieving fast AOTC-to-AOTC calls. We generate another C function for the AOTC method, which takes the Java operand stack as an argument and makes a call to the original AOTC C function with the individual operand stack slots as actual arguments. This wrapper function will be invoked by an interpreted caller, so the interpreter-to-AOTC calls are made by two function calls. The code sequences for an AOTC caller, an interpreted caller and an AOTC callee method are shown in Fig. 7 (this double-call overhead can be reduced by replacing the second call by a direct jump which will make the return from the callee_method() return to the caller of the wrapper_callee() directly).

The reason why this call interface leads to a faster interpreter-to-AOTC call compared with the standard C interface is that we can specialize the argument copying routine of the standard C interface (the assembly routine in Fig. 4b) according to the individual AOTC callee method, instead of sharing it among all AOTC callee methods; that is, the overhead of loop iterations and runtime checks of argument or return types in the argument copying routine can be obviated in the wrapper function since we can determine the exact number and the type of arguments as well as the return type when we generate it (by consulting the original AOTC C function). So, the arguments for the call made in the wrapper function can be appropriately prepared based on

---

**FIGURE 6.** An example of the mixed call interface.
what are expected in the original C function. It should be noted that the standard C interface also requires two calls to reach the original AOTC C function (a call to the assembly routine followed by a call to the AOTC function) as in this interface.

3.3. AOTC optimizations

One of the most important benefits of using b-to-C AOTC is that we can resort to the existing, powerful optimizations of gcc. However, gcc optimizations are not enough since they cannot exploit Java-specific optimization opportunities, which should be handled by the AOTC directly. In this section, we discuss such Java-specific optimizations performed by our AOTC.

3.3.1. Null pointer check elimination

JVM specification requires checking if an object pointer is NULL before referencing it. Therefore, our AOTC adds a check code right before each object reference. Obviously, many of these checks are redundant or unnecessary, so we need to remove them. Since it is relatively simple to confirm if a pointer is not null to remove unnecessary checks, existing C compiler optimizations may eliminate them. In fact, we found that gcc includes such optimizations. However, since the AOTC can understand the control flow information better than gcc (e.g. gcc would regard the check code as a regular branch, which is hard to remove) and can know that some reference is never null (e.g. the this pointer), AOTC can remove checks better than gcc.

Figure 8 shows an example of such optimizations. The bytecode aload_0 and getfield_4 in Fig. 8a access a field based on the this pointer within a method, which is translated to Fig. 8b where we need to check if the pointer C variable s0_ref is NULL. However, the this pointer in a method can never be null, so such a check is unnecessary. The optimizations in gcc cannot exploit this information and remove the check.

3.3.2. ABC elimination

An array access in Java must be assured that the array index is not out of bounds. Therefore, our AOTC should generate a C statement for an ABC for each array access bytecode, which would slow down the program execution seriously. Unlike the null pointer check, it is not simple for gcc to remove redundant and unnecessary ABCs since such an optimization is much more complicated. In fact, gcc cannot even distinguish a pointer to an array from regular pointers.

We employed the ABC analysis technique proposed by Gupta [37] and adapted it for use in the context of Java, especially for satisfying Java’s precise exception semantics. The adapted algorithm performs a backward analysis to identify
redundant ABCs first, followed by a forward analysis to actually remove them after modifying remaining ABCs if needed. One complication is that we should not violate the precise exception semantics, so that we do not propagate the ABC information during the backward analysis beyond bytecodes whose execution can change the program states. We also remove redundant negative array size checks.

Figure 9 shows an example of our ABC elimination. Figure 9a is the bytecode that reads two array elements a[i] and a[i+1]. We first read a[i] based on the variable 1 which is a pointer to the array a, and the variable 2 which is the index i. Then, we read a[i+1] based on the variable 1 and the variable 2, added by one. Figure 9b is the translated C code where there are two ABCs, check[0 <= L2_int < L1_ref.length] and check [0 <= L2_int + 1 < L1_ref.length] (actually the ABC is implemented by a single comparison with unsigned typecasting [41], which obviates the negative array index check; the first ABC is implemented by check[(unsigned)L2_int < L1_ref.length], for example). The second ABC can be eliminated, if we replace the first ABC by a stronger one that covers the second ABC as well. Figure 9c shows the result where the backward analysis detects the partial overlap of both checks and the forward analysis removes the second check after updating the first one (i.e. subtract one from L1_ref.length in the inequality check condition).

One thing to note that if the first array access were a write, not a read, the optimization cannot ensure the precise exception semantics of Java. For example, if the access of a[i+1] is out of bounds, then the original program in Fig. 9b will generate an exception at the second ABC, while the optimized program in Fig. 9c will generate an exception at the first ABC. The problem is that the original program will pass an array modified by the write to the exception handler while the optimized program will pass an unmodified array, leading to a different exception behavior. Therefore, the upper bound check of the second ABC cannot be removed completely and only the lower bound check can be removed; unfortunately, since both the upper and lower bound check is implemented by a single check after unsigned typecasting, we cannot remove the check, and so the ABC remains. Our modified analysis technique can detect this and guarantee a precise exception behavior. In fact, if there is any semantic-violating bytecode (e.g. local variable write, putfield, putstatic, invoke, exception check, etc.) between two ABCs, we cannot remove the ABC. Unfortunately, this restriction causes many ABC opportunities to be lost, which limits the performance impact (see Section 4).

3.3.3. Method inlining

Java as an object-oriented language includes many calls of short methods, and so the call overhead takes a significant portion of the total execution time. Therefore, our AOTC should perform aggressive method inlining, as most Java JITCs do.

Most optimizing compilers including gcc already perform function inlining. However, gcc inlining in the context of the translated C program is not enough since it cannot perform any Java-specific inlining; that is, gcc would perform inlining only for calls to a C function corresponding to a short, static Java method, because gcc normally does not perform inlining for large functions and it can know the call target only for the functions of static methods; that is, the translated C code for a virtual method call accesses the method table and the method block to obtain the pointer of the target function (see Fig. 7), and so the call target can be known only at runtime, which makes the gcc inlining impossible. In fact, when we turn on the gcc inlining flag, we do not notice any performance improvement (and the code size increase was also small), indicating little opportunities for gcc inlining. Therefore, we perform inlining in the context of the AOTC, which leads to a significant performance impact, as will be seen in Section 4.

We perform inlining for static and final methods easily since the callee is known at translation time. We attempt to inline small-sized methods only since excessive code expansion due to inlining can affect the performance negatively. With profiling information, however, we even inline large, hot methods. For virtual methods, we perform inlining based on the profiling
information, but we add a check code to confirm if the inlined method is really the right method to call, and if it fails, we jump to the interpreter routine to find the correct target method.

3.3.4. Profiling
To achieve high performance, it is desirable to profile the Java middleware so that the profiled information can be used during its AOTC. We actually perform such profiling off-line, especially for counting method calls and loop iterations. This can be useful in choosing AOTC candidates, if the memory constraints of embedded systems do not allow compiling all methods, as well as in performing feedback-oriented optimizations.

3.3.5. Copy propagation of references
As a stack machine, JVM requires many pushes and pops between the operand stack and the local variables, which are translated as copies between stack C variables and local C variables. Most of these copies can be removed later by gcc, but there are some issues related to the copies of object references. To support precise GC, our AOTC generates an additional C statement that saves a reference C variable to the Java stack frame whenever a reference C variable is updated. So a copy statement that updates a reference C variable also requires such a frame save (e.g. S0_ref=L1_ref; is followed by frame[0]=S0_ref;). The issue is that the frame save remains even if the copy is removed later by gcc. To reduce these unnecessary frame saves, our AOTC performs copy propagation during the translation, which removes such copies, thus obviating these frame saves [33].

3.4. Some compatibility issues
It is required that the b-to-C translation keeps the semantics of Java, and there can be some compatibility issues, especially related to the different behavior of arithmetic calculation between Java and C. For example, the integer division bytecode idiv is supposed to produce −2147483648 (0x80000000) when dividing −2147483648 by −1, while it often overflows in C. So, if we translate idiv directly into a C integer division, the behavior would be different. The JVM interpreter handles the arithmetic using a special macro, and our b-to-C AOTC generates C code that uses the macro instead of C integer division, which leads to the same result as the interpreter. For example idiv is translated to a call to the following macro, with the operator equaled to '/':

```c
#define intBinaryOp2(op1, op2, operator)
((op1)==0x80000000 && (op2)==−1) ? ((op1) operator 1) : ((op1) operator (op2))
```

The C compiler optimizations might also lead to a different behavior. If there is a Java expression if (x < x + 1) and if x + 1 overflows, Java evaluates the condition as false. If we translate it directly to the same C expression, the C compiler would often evaluate the condition as true after optimization (we confirmed it in the case of the gcc). However, the bytecode sequence for the Java expression is translated by our AOTC to the following C code sequence: S1_int = S0_int + 1; if (S0_int < S1_int). In this case, we found that the condition is evaluated as false by the gcc, so that we can have the same result (It appears that the gcc does not perform such optimizations beyond a single statement). A more definite method is providing the −fwrapv flag when compiling with the gcc, which avoids undefined behavior, thus disabling those optimizations that can lead to undefined results.

Many of the original JVM features or APIs are still supported by our AOTC because they are often implemented by JVM functions or macros, so our AOTC can generate C code that simply calls them. For example most of the thread APIs are native methods, hence not being compiled by our AOTC. For synchronized methods or blocks, the monitorenter and monitorenter bytecode for acquiring and releasing the lock are translated to corresponding function calls in the JVM (CVMObjectLock() and CVMObjectUnlock(), respectively). For the scheduling of the currently executing threads, a macro call is generated to check if there are any waiting threads (i.e. CVMthreadSchedHook()) at the loop backedge and at the prolog of the translated code.

Similarly, most of the reflection and the security APIs are implemented by native methods, and so they are not handled by the AOTC. For the security APIs, however, there is a correctness issue related to AOTC. For the permission management, the security API methods such as getCallerClassLoader() or getCallerClassLoader() inspect the Java stack for retrieving the class information or the class loader information for the methods in the call stack. The problem is that for the AOTC methods, the Java stack frames are not allocated as in the interpreted methods, so if AOTC methods are on the call stack when those API methods are called, incorrect information can be retrieved. Fortunately, we found that those API methods need to inspect either their caller method frame or their caller’s caller method frame only in our JVM system libraries. This means that we can simply disable AOTC for a method if it calls those API methods or its callee calls those API methods, which can avoid the problem. For example we disable AOTC for forName() and newInstance() in the Class class and doPrivileged() in the AccessController class so that they are always interpreted.

Input-dependent dynamic class loading can still work with our AOTC since newly loaded classes will be executed by the interpreter. So the performance can be affected by dynamic class loading, yet the correctness is still preserved even if profile-based optimizations were made without new classes.

4. EXPERIMENTAL RESULTS
The previous section described the AOTC design issues including compilation methods, call interfaces, and optimizations, and
showed the choices implemented in our AOTC. In this section we evaluate them, compared with other alternatives.

4.1. Experimental environment

We experimented with Sun’s CVM Reference Implementation for which we implemented the b-to-C AOTC. The experiments were performed on a MIPS-based SoC called AMD Xilleon, which is popularly employed in DTVs. The MIPS CPU has a clock speed of 300MHz and has a 16KB L1-cache/16KB D-cache, with a 128MB main memory. The OS is Linux with kernel 2.4. The translated C code is compiled by GNU MIPS Cross C Compiler version 3.2.1, with an optimization flag –O3 (some experiments were performed with –O0 in order to isolate gcc’s optimizations; see Section 4.5). The benchmarks we used are the EEMBC [38] and some of the SPECjvm98 benchmarks [39]. We ran each benchmark five times and took the minimum running time (we found that the variation of the running time for those five runs is <1%). Both the interpreter and the AOTC were experimented with a romized environment where all classes are romized by Sun’s romizer called the JavaCodeCompactor, which performs constant pool resolution in advance as well as the romization.

4.2. AOTC performance

Figure 10 depicts the b-to-C AOTC performance compared with the interpreter performance when all executed methods are AOTCed. This AOTC is based on combined compilation and the wrapper function interface, with all Java-specific optimizations enabled. The graph shows that the AOTC achieves a geometric mean of 4× performance compared with the interpreter (our JITC implemented on the same platform achieves a geometric mean of 2.7× performance over the interpreter [31]).

The performance advantage of AOTC compared with the interpreter or the JITC is something expected and is not a main issue of this paper (it can actually vary depending on the VM implementation). The main focus of this paper is the performance impact of individual features and optimizations of the AOTC, which will be reported next.

4.3. Combined vs. separate compilation

We first attempt to evaluate our combined compilation, compared with separate compilation. Since we could not actually develop a new AOTC based on separate compilation, we experimented as follows. We performed AOTC based on combined compilation but added an indirection table such that all out-of-class method calls or field accesses are made indirectly through the table as discussed in Section 3.1.1. For example an out-of-class static method call is translated to

\[
S0_{\text{int}}=\text{table[callee_index]}(\text{ee},S0_{\text{int}},S1_{\text{int}})
\]

instead of

\[
S0_{\text{int}}=\text{callee_method}(\text{ee},S0_{\text{int}},S1_{\text{int}})
\]

The entries of the indirection table are filled at the JVM boot time such that the matching addresses and offsets for all out-of-class calls and accesses are found through an exhaustive search. So, if we measure the running time after the JVM boot, it will precisely include the access overhead of the indirection table in separate compilation. We also disallow inlining across different classes during AOTC, and so the impact of no inlining will also be included.

Figure 11 shows the performance of this version of separate compilation compared with that of combined compilation as 100%. We also include the performance of combined compilation when no out-of-class inlining is allowed, in order to understand the impact of no inlining across classes. The graph shows that separate compilation is associated with a serious performance loss, most of which comes from no inlining, yet the overhead of indirection table access is also tangible. Actually, we analyzed the assembly code generated from the translated C code, and found that the access of the indirection table is associated with not only a load from the table, but a copy and a nop, which seem to be unnecessary. This appears to be some gcc optimization problem.

We also measured the boot time overhead caused by filling the indirection table, which would mimic the overhead of runtime resolution and linking. For simplicity we actually built one big
4.4. Evaluation of call interfaces

We also evaluated the four call interfaces: the standard C call interface, operand stack call interface, mixed call interface and wrapper function interface. We performed the evaluation in two environments, the full-AOTC environment where all executed methods are compiled by the AOTC, and the partial-AOTC environment where the library methods are compiled by the AOTC while the application methods are executed by the interpreter (library classes and application classes are modeled as the middleware and the downloaded classes, respectively).

By experimenting in both environments, we can evaluate the performance of the interpreter-to-AOTC calls as well as the AOTC-to-AOTC calls.

Figure 12 shows the performance result in the full-AOTC environment. The standard C call interface is depicted as a basis of 100%, compared with the other three interfaces. It shows that the standard C call interface is the fastest in the full-AOTC, which would be due to the lack of copying arguments to and from the operand stack in the operand stack interface, or due to the lack of checking if the caller is an AOTC method or an interpreted method in the mixed call interface. The wrapper function interface follows the standard C call interface for AOTC-to-AOTC calls, thus showing a similar, competitive performance. There are some benchmarks where the wrapper function interface outperforms the standard C call interface (e.g. Compress, Db, Mpegaudio). We do not know the exact reason for this, but it might be related to those methods that are still interpreted because of the compatibility issues discussed in Section 3.4, so there are interpreter-to-AOTC calls where the wrapper function interface is advantageous. Also, some virtual method calls whose target check failed must jump to the interpreter routine for finding the correct target method as discussed in Section 3.3.3 and then jump to the target AOTC method, where the wrapper function interface leads to less overhead than the standard C call interface.
Figure 13 shows the performance result in the partial-AOTC environment with the standard C call interface as a basis of 100%. Here, the performance of the operand stack call interface is superior to the standard C call interface. This is due to the frequent calls from interpreted methods to AOTC methods where the overhead of copying arguments by the interpreter to the registers or the C stack is substantial, especially due to loop iterations and argument type testing routines shared by all callee methods. Both the mixed call interface and the wrapper function interface show a performance similar to the operand stack interface. Using a wrapper function reduces the overhead of the shared routine, even though it is based on the standard C interface. Consequently, the wrapper function interface is competitive for both the AOTC-to-AOTC calls and the interpreter-to-AOTC calls.

4.5. Java-specific optimizations

We evaluated the Java-specific optimizations by the AOTC, including null-pointer check elimination, ABC elimination, method inlining (with profiling) and copy propagation. To isolate the performance impact of these optimizations from the gcc optimizations, we first experimented with the –O0 flag (no optimization) of gcc. Figure 14 shows the full-AOTC performance with all Java-specific optimizations turned on, compared to that only with gcc –O0 as a basis of 100%. Our
Java-specific optimizations lead to a performance improvement of an average of 52%.

Figure 15 shows the performance impact of individual optimizations, including null-check elimination, ABC elimination and copy propagation, compared to the gcc –O0. It shows that copy propagation of references is the most effective (an average of 19%), which is due to elimination of frame saves due to the copy removal. Null-check eliminations and ABC elimination lead to an average performance improvement of 7 and 2%, respectively.

Figure 16 complements the result of Fig. 15 by showing the dynamic amount of the code removed by each elimination technique. Around 61% of the null-check code and around 39% of the reference copies are removed. On the other hand, ABCs are not removed tangibly, except for Crypto, Parallel and Mpegaudio. We found that this is due to the precise exception semantics of Java which keeps many candidate ABCs from being removed. Performance decreases tangibly in PNG and increases tangibly in Crypto and RegEx, while ABCs are not removed much in those benchmarks. We do not know the exact reason for these performance anomalies, but gcc optimizations appear to affect the code quality significantly after the ABC removal.

Figure 17 shows the performance impact of method inlining, with no profile feedback and with a profile feedback, respectively. With no profile-feedback, there can be some
performance degradation as in PNG and DB, since useless inlining can affect the performance negatively due to the increased code size. With a profile feedback, however, we can obtain a consistent and tangible performance improvement as shown in Fig. 17, which is an average of 13%.

Figure 18 shows the percentage of dynamic method calls removed due to inlining. We can see that many method calls are removed in PNG and RegEx when inlined with profile feedback, and this led to a significant performance impact in Fig. 17. For compress, method calls are removed significantly, both with and without profile feedback, and hence its tangible performance improvement. On the other hand, parallel shows little performance impact even though many method calls are removed. This is due to the small number of method calls whose removal cannot affect the performance tangibly.

4.6. C Optimization flags

We also evaluated the Java-specific optimizations with the gcc optimizations turned on. Before we do this evaluation, we first measured the performance of our AOTC with –O0 (no optimizations), –O2 (standard optimizations) and –O3 (O2 plus function inlining and renaming), which are depicted in Fig. 19. The –O3 optimization increases the –O0 performance by 2.1 times.
Figure 20 shows the performance impact of Java-specific optimizations, when we compile the translated code with gcc –O3. It shows that the average performance impact is 41%, which is lower than the case of gcc –O0 in Fig. 14. This is obviously due to the optimizations performed by gcc, which reduces the impact of Java-specific optimizations.

Figure 21 shows the individual performance impact of null-check elimination, ABC elimination and copy propagation with the –O3 flag. It shows that copy propagation of references is still the most effective (an average performance improvement of 13%), while null-check eliminations and ABC elimination lead to average improvements of 1 and 3%, respectively. These improvements are lower than those with the –O0 flag in Fig. 15. One thing to note is that gcc can eliminate null checks better than ABCs since gcc cannot detect array objects. This is the reason why the performance impact of null check elimination is smaller than that of ABC elimination with –O3 in Fig. 21, unlike with –O0 in Fig. 15. For the ABC elimination, Fig. 21 shows a slightly better performance impact (3%) than in Fig. 15(2%). This is due to the reduced total running time obtained with –O3, which makes the performance impact of the ABC elimination slightly more pronounced than with –O2.

Figure 22 shows the performance impact of method inlining, with no profile feedback and with a profile feedback, respectively. We can observe that the performance impact of inlining with gcc –O3 is an average of 20%, which is higher than that with gcc –O0 in Fig. 17. The reason is that the inlining with gcc –O0 will simply reduce the call overhead only, while inlining with gcc –O3 will perform optimizations for the inlined function as well, which would increase its performance impact.

### 4.7. Static memory overhead

Translation of bytecode into machine code is involved with a static memory overhead due to an increase of the object code size. Figure 23 shows the static memory requirement of the (full) AOTC-mode of executions (one with translating all methods and the other with translating executed methods only), compared with that of the interpreter-mode of execution (no translation) as 100%, for EEMBC and SPECjvm98 benchmarks. Each bar includes the static memory size of the JVM executable at the bottom, so that we can distinguish the static memory size for the benchmark programs separately.

The first bar is for the interpreter-mode execution, where the static memory size for the romized class files for all of the system classes and application classes is depicted. Romization of class files means converting them directly into C data structures as they are and compiling them together with the JVM. Therefore, Java methods are kept as the bytecode programs in the JVM.
executable. The next bar is for the AOTC-mode when we compile all system and application methods. The third bar is for the AOTC-mode when we compile only executed system and application methods, such that the unexecuted methods are romized into the bytecode programs, thus reducing the static memory overhead.

Figure 23 indicates that the static memory overhead for the AOTC of all methods is substantial (250% for EEMBC and 300% for SPECjvm98), while the overhead for the AOTC of executed methods only is relatively reasonable (40% for EEMBC and 100% for SPECjvm98). In fact, we can know most of the executed methods through profiling before we perform AOTC, achieving the latter.

4.8. Dynamic memory overhead

Execution of the translated machine code may also be involved with a dynamic memory overhead due to the increased code space, causing a larger footprint. Figure 24 depicts the peak memory footprint measured during the interpreter-mode execution and the AOTC-mode execution for each benchmark. We measured it using the top command in Linux, taking the peak value for the RSS (the total amount of physical memory used by the task) of the JVM executable. It shows an average additional footprint of 15%.

4.9. Comparison to HotSpot virtual machine

To evaluate the performance of our b-to-C AOTC, we compared it against a high-performance virtual machine, the HotSpot VM from Oracle. We made the comparison in two platforms, a real DTV and a high-performance desktop PC. Our experimental DTV platform has a 333 MHz MIPS CPU with a 128 MB memory. Its software platform has an open source version of Sun’s CDC HotSpot VM, called the phoneMe Advanced (MR2 version) [27], running on the Linux kernel 2.6.12. In this platform, the C code generated by our AOTC is compiled by GNU MIPS Cross C Compiler version 3.4.6. For the HotSpot VM, we made two experiments, a cold run and a warm run. The cold run is simply running each benchmark once and measuring its running time. The warm run is running each benchmark five times in sequence and measuring the running time of the last run, which would obviate the JITC overhead and get most of the hot and warm methods executed in machine code (we found that after three times of execution, there is little difference of running time).

Figure 25 shows the performances of the PhoneME Advanced interpreter (100%), the HotSpot-cold, the HotSpot-warm and our AOTC. The HotSpot-cold is six times faster than the interpreter, while the HotSpot-warm is seven times faster. Our AOTC is four times faster, which means a 30 and 40% slower performance than the HotSpot-cold and the HotSpot-warm.
respectively. Other b-to-C AOTCs for embedded systems such as OVM [11] or Jamaica [13] are reported to be 50% slower than Oracle’s high-performance VMs (see Section 5).

The desktop PC environment is a dual-core 3.2 GHz Pentium 4 with a 2G memory and the OS is Linux kernel 2.6.28. It has Oracle’s HotSpot Server 1.6, and the compiler used for our AOTC is gcc 4.3.3. We also experimented with two modes for the HotSpot Server, HotSpot-cold and HotSpot-warm. Figure 26 shows the performances of the HotSpot Server interpreter (100%), the HotSpot-cold, the HotSpot-warm and our AOTC. It shows that the HotSpot Server JITC is faster than the interpreter by five times (HotSpot-cold) and 19.7 times (HotSpot-warm), and the difference is much bigger than in the DTV platform. It appears that the JITC overhead of the HotSpot Server is much higher than the PhoneMe Advanced, yet the quality of the generated code is much better. Our b-to-C AOTC achieves a performance benefit of 9.7 times, which is lower than the HotSpot-cold but higher than the HotSpot-warm. HotSpot-warm appears to perform many Java-specific optimizations. Our AOTC outperforms the HotSpot-cold unlike in the DTV, and this seems to be related to the gcc version and the target CPU, such that the gcc 4.x version for the x86 generates much better code than the gcc 3.x version for the MIPS CPU. Generally, we found that AOTC-generated code has a much larger prolog and epilog code than JITC-generated code, because AOTC should generate them for both Java stack and native stack, while the JITC generates them for Java stack only. We believe that this is one reason for the lower performance of AOTC compared with JITC.

5. RELATED WORK

In this section, we describe the previous b-to-C AOTCs, in terms of the compilation strategy, the call interface and the Java-specific optimizations evaluated in this paper. However, many of the previous AOTCs do not deal with these issues specifically. Also, they do not support precise GC or fast exception handling, and often target an incomplete JVM where some of the main JVM features such as threads, reflections, or dynamic class loading are not supported, unlike the b-to-C AOTC used in our evaluation.

5.1. Generic b-to-C AOTC

Toba performs b-to-C AOTC for all used methods and compiles them together with the JVM into a single executable, as in our AOTC [5]; the AOTC-to-AOTC calls therein follow the standard C interface. Unfortunately, its AOTC methods cannot work with the interpreted methods, thus not working for a partial-AOTC environment where downloaded classes exist. As to the optimizations, it does not perform any Java-specific optimizations.

Harissa scans through the bytecode stream starting from the main method in order to find all of the possibly used methods and performs a b-to-C AOTC for them [6]. Its AOTC-to-AOTC call interface is also based on the standard C interface, and so the interpreter-to-AOTC calls are associated with a high overhead. Harissa performs inlining and ABC elimination, but it does not perform copy elimination and profile-based optimization.

Java-thru-C performs AOTC only for used methods, fields and classes [7]. For optimizations, it employs soot, a Java bytecode optimization framework [40], which performs method inlining, copy propagation and other traditional optimizations, after converting the bytecode to an intermediate representation called the jimple code. The optimized jimple code is then translated to the C code. No Java-specific optimizations such as redundant code elimination are performed. Also, its AOTC methods cannot work with interpreted methods.

5.2. Real-time b-to-C AOTC

Many hard real-time JVMs employ the AOTC since the JITC may cause non-deterministic behavior since it runs during the execution of the real-time application at unpredictable times, while the interpreter is too slow for meeting the real-time constraint [8–17]. Unfortunately, they provide little detail on the design and implementation of the AOTC itself but focus more on real-time GC or real-time lock synchronization.
A real-time Java AOTC called the Lund Java-based Real-Time (LJRT) provides a research prototype compiler for the Java-to-C translation [8], whose main goal is to provide hard real-time guarantees while using the standard Java memory model with one heap and a real-time GC. LJRT searches for all classes starting from the main class, and parses them into an abstract syntax tree (AST). The AST is first transformed to a form suitable for code generation and for easier use of the GC interface proposed in their work, and then is translated to C code.

Ovm is a framework for building language runtimes with a b-to-C AOTC where the compilation to machine code is done by gcc [9–11]. For a given Java application it compiles them as a whole, producing a single stand-alone executable as ours. It does not allow dynamic class loading. These features allow more aggressive optimizations, leading to high-performance code. Unfortunately, they did not report the performance impact of individual optimizations, except for a brief description of inlining and copy propagation. They evaluate their AOTC in comparison with GCJ and Oracle’s JDK HotSpot Server/Client. Their AOTC outperforms GCJ, but achieves a performance less than half of those of the HotSpot Server/Client.

For experimenting with real-time GCs, the Jamaica VM was introduced [12]. It includes a b-to-C compiler that compiles the translated C code and the JVM C code into a single executable. Although some description of its compiler structure is included in [13], it is not as detailed and comprehensive as ours. The Jamaica VM is reported to show a lower performance than Oracle’s JDK 1.2.2, by more than 50% except for compress and db.

IBM WebSphere Real-time VM has been used commercially, and it includes a component called the ahead-of-time (AOT) compiler [14]. The AOT compiler can produce high-performance static code with ample optimizations since the compilation overhead is not part of the running time unlike the JITC, yet it conservatively disallows some useful optimizations in order to meet the JVM features such as dynamic class loading. For example it disallows method inlining since the inlined method can be invalid later if it is reloaded dynamically.

PERC pico [15] is a commercial JVM product for safety-critical Java technology, which employs the AOTC. Unfortunately, the details of the AOTC implementation are not provided.

A real-time JVM called Fiji [16, 17] includes a b-to-C AOTC where the compiled machine code from the translated C code is linked with the Fiji runtime. The Fiji AOTC includes many optimizations such as virtualization, devirtualization, inlining, copy propagation, sparse conditional constant propagation, tail duplication, type propagation, null check elimination, ABC elimination, global value numbering, load-load optimization, loop peeling and unrolling. However, they reported only the correlation of the dynamic count of null checks, ABCs and type checks with the overall running time, without the full analysis of the individual optimizations. They evaluated Fiji in comparison with Oracle’s JDK HotSpot and IBM WebSphere.

With regard to the CDj benchmark, which is for measuring the worst-case performance, Fiji outperforms other VMs [16, 17]. For SPECjvm98, Fiji performs around 16% worse than HotSpot 1.6 Server [17], which is reasonably good.

We should mention that the performance of hard real-time AOTCs described above should be interpreted carefully in order to avoid any confusion, especially when compared with our embedded AOTC results described in Section 4.9. First, it is not completely fair to compare a real-time VM with a regular VM since the primary goal of a real-time VM is not achieving high performance but meeting the worst-case deadlines. Secondly, most real-time AOTCs mentioned above can work on server-level systems, possibly with multiprocessors, while the JVM used in our experiment is primarily for single-CPU embedded systems. Finally, real-time VMs often target a ‘closed-world’ system without dynamic class loading, while regular VMs including our AOTC supports an ‘open-world’ system. This closed-world assumption allows obviating many type checks through the whole program analysis (e.g. our inlining safety checks in Section 3.3.3 can be omitted) and leads to additional optimization opportunities, which would be advantageous for performance. The performance of Fiji described above, for example, is with this closed-world option enabled, and would drop if it is disabled (no open-world performance for Fiji has been reported in the literature, though). The performance of our AOTC is obtained with the open-world assumption, and so we believe it is competitive.

5.3. Miscellaneous

GCJ compiles Java programs or Java bytecode into machine code using the gcc with its modified front-ends [18], and the machine code is interoperable with its interpreter, GIJ. Since GCJ can compile the Java source code to the machine code directly, without passing through the bytecode as in our b-to-C AOTC, there can be some advantage in optimizations. However, GCJ does not employ any profile-based optimizations such as inlining of virtual calls. Also, GCJ supports the standard C call interface only, and so there can be some performance disadvantage in a hybrid environment where the GCJ-generated code interoperates with GIJ. Finally, GCJ does not support JNI or CNI precisely, and so there is a chance that it cannot support many of the Java APIs correctly.

Instead of fully translating the Java code to C code, Java-to-C bridge proposes a new intermediate language where Java code and C code can be used in a mixed way [42]. The main goal is not to increase the performance but to improve the productivity of the programmers by allowing them to use both Java code and C code for the same program, without the complexity of using JNI.

Our AOTC is not based on JNI in the sense that our C functions can access the Java objects and JVM data structures or functions directly, instead of accessing them through predefined JNI functions for string handling, array handling, method
handling, member variable handling, exception handling, etc., as JNI C functions are supposed to do. Therefore, we can avoid a serious performance penalty associated with accessing through those JNI functions. We follow only the JNI’s function naming convention, and as such, we follow a similar implementation of the JNI method handling functions for interpreter-to-AOTC calls.

6. SUMMARY

The b-to-C AOTC is one of the most promising approaches to embedded Java acceleration, especially for the Java middleware and system classes installed on the client devices, since we can generate highly optimized machine code for them in an off-line fashion. This paper discussed various design and optimization alternatives for AOTC and evaluated them. First, we presented two approaches to compiling the translated C code, separate compilation and combined compilation. Separate compilation is advantageous when there is a frequent update for the middleware, although it incurs a relatively small performance penalty compared with combined compilation. Secondly, we presented four call interfaces for efficient handling of the interpreter-to-AOTC calls as well as the AOTC-to-AOTC calls. Our evaluation results show that the wrapper function interface leads to the best performance for both types of calls. Finally, we proposed Java-specific optimizations to complement the gcc optimization, which shows a substantial performance advantage according to our evaluation. We also found that the static and dynamic memory overhead of the AOTC is relatively reasonable, considering its performance benefit. These results will be helpful to anyone who wants to build a b-to-C AOTC for the embedded Java platform.

FUNDING

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-313-D00628).

REFERENCES

[27] java.net the source for java technology collaboration. Sun Microsystems, Santa Clara, CA, USA. http://wiki.java.net/bin/view/PhoneMEAdvanced/PhoneMEAdvanced.