Journal of Circuits, Systems, and Computers
Vol. 21, No. 2 (2012) 1240001 (22 pages)

DOI: 10.1142/S0218126612400014

World Scientific
www.worldscientific.com

HYBRID JAVA COMPILATION OF JUST-IN-TIME AND AHEAD-OF TIME FOR EMBEDDED SYSTEMS*

HYEONG-SEOK OH†, SOO-MOOK MOON‡ and DONG-HEON JUNG§

School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 151-742, Korea
†*oracle@altair.snu.ac.kr*
‡*smoon@altair.snu.ac.kr*
§*clamp@altair.snu.ac.kr*

Received 25 July 2010
Accepted 23 January 2011

Many embedded Java software platforms execute two types of Java classes: those installed statically on the client device and those downloaded dynamically from service providers at runtime. For higher performance, it would be desirable to compile static Java classes by ahead-of-time compiler (AOTC) and to handle dynamically downloaded classes by just-in-time compiler (JITC), providing a hybrid compilation environment. This paper proposes a hybrid Java compilation approach and performs an initial case study with a hybrid environment, which is constructed simply by merging an existing AOTC and a JITC for the same Java virtual machine. Both compilers are developed independently for their own performance advantages with a generally accepted approach of compilation, but we merged them as efficiently as possible. Contrary to our expectations, the hybrid environment does not deliver a performance, in-between of full-JITC's and full-AOTC's. In fact, its performance is even lower than full-JITC's for many benchmarks. We analyzed the result and found that a naive merge of JITC and AOTC may result in inefficiencies, especially due to calls between JITC methods and AOTC methods. Based on these observations, we propose some ideas to reduce such a call overhead. We also observed that the distribution of JITC methods and AOTC methods is also important, and experimented with various distributions to understand when a hybrid environment can deliver a desired performance. Finally, we discuss how JITC and AOTC should be designed for efficient hybrid execution.

Keywords: Hybrid Java compilation; ahead-of-time compiler; just-in-time compiler; JVM.

1. Introduction

Java is a popular software platform for many embedded systems, including digital TVs, mobile phones, or bluray disks. This is mainly due to its advantage in platform

*This is a revised and expanded version of a paper published in the 12th Workshop on Interaction between Compilers and Computer Architectures (Interact-12), Salt Lake City, Utah, USA (Feb 16, 2008). This paper was recommended by Regional Editor Xin Yuan.
‡,§Corresponding authors.

independence, security, and software development. That is, the use of a virtual machine allows a consistent runtime environment for diverse client devices that have different CPUs, OS, and hardware components. Moreover, Java has little security issues such that it is extremely difficult for malicious Java code to break down a whole system. Finally, it is much easier to develop software with Java due to its sufficient, mature APIs and its robust language features such as exception handling and garbage collection.

The advantage of platform independence is achieved by using the Java virtual machine (JVM) that executes Java's compiled executable called the bytecode. The bytecode is a stack-based instruction set which can be executed by an interpreter on any platform without porting the original source code. Since this software-based execution is obviously much slower than hardware-based execution, compilation techniques for translating bytecode into machine code have been used, such as just-in-time compilers (JITC)[1] and ahead-of-time compilers (AOTC).[2–5] In embedded systems, JITC performs an online translation on the client device at runtime, while AOTC performs an offline translation on the server before runtime and the translated machine code is installed on the client device.

Generally, AOTC is more advantageous in embedded systems since it obviates the runtime translation overhead of JITC, which would waste the limited computing power and runtime memory of embedded systems, and may affect the real time behavior of the client devices. On the other hand, many embedded systems such as digital TVs, mobile phones, and bluray disks may download classes dynamically at runtime, which cannot be handled by AOTC and thus, should be executed by the interpreter. However, the performance benefit achieved by AOTC can be easily offset by such interpretive execution. So it would be desirable to employ JITC as well to handle dynamically loaded classes for complementing AOTC. That is, we need a hybrid compilation environment.

We actually constructed such a hybrid environment by merging an AOTC[6–8] and a JITC,[9] each of which takes the most generally accepted approach of compilation, commonly used by others. We evaluated the environment with an experimental setup for hybrid compilation. Although both the AOTC and the JITC were developed independently for their own performance advantages, we merged them as efficiently as possible to reduce any overhead of hybrid execution. Consequently, our experiences with this environment can help those who want to build their own hybrid environment and warn them of any possible inefficiencies. We also discuss how JITC and AOTC should be designed for efficient hybrid execution, which will provide a useful insight on their interface problem. These are the main contributions of this paper.

The rest of this paper is organized as follows. Section 2 introduces the approach of hybrid compilation environment and motivates readers by presenting an initial evaluation result with our hybrid environment. Section 3 reviews the JITC and the AOTC used for the hybrid environment and Sec. 4 describes how they are merged. Section 5 includes our experimental results and analyzes with the environment. It also

investigates how we could possibly achieve a better hybrid performance. Comparison to the previous work is in Sec. 6. A summary and a future work follow in Sec. 7.

2. The Approach of Hybrid Compilation and an Initial Case Study

Our proposed hybrid compilation environment targets an embedded Java software platform which can download classes at runtime. For example, a software platform for digital TVs (DTV) is typically composed of two components: a Java middleware called OCAP or ACAP (and Java system classes) which are statically installed on the DTV set-top box, and Java classes called xlets which are dynamically downloaded through the cable line or the antenna. Also, a software platform for mobile phones is composed of the MIDP middleware on the phone and midlets downloaded via the wireless network. Bluray disks consist of the BD-J middleware on the BD player and xlets on the BD titles. We believe these dual-component systems will be a mainstream trend for embedded Java software architecture.

Another trend of these dual-component systems is that both the Java middleware and the downloaded classes become more complex and substantial. The initial downloaded Java classes were mainly for displaying idle screen images or for delivering simple contents, but now more substantial Java classes such as games or interactive information that take a longer execution time are being downloaded. In order to reduce the network bandwidth (wired or wireless) for downloading, the Java middleware also gets more substantial to absorb the size and the complexity of downloaded classes. In mobile phones, for example, the first MIDP middleware provided libraries for user interfaces only, yet its successor middleware called the JTWI provided an integrated library for the music players and the SMS as well. Now a more substantial middleware called the MSA with more features is being introduced.

For achieving high performance on these substantial, dual-component systems, it would be desirable to employ hybrid acceleration such that the Java middleware is compiled by AOTC while the downloaded classes are handled by JITC. However, our point is that a naïve merge of an existing AOTC and a JITC would not lead to a performance level that we would normally expect from a hybrid environment. In order to motivate readers, we actually constructed a hybrid environment by merging an AOTC and a JITC, which we developed independently for the same JVM, and then we experimented with it as follows.

Both the AOTC and the JITC targets Sun's CDC VM (CVM). The AOTC takes a bytecode-to-C approach such that the bytecode is translated into C code, which is then compiled with the CVM source code using a GNU C compiler. The JITC uses adaptive compilation method, where Java methods are initially executed by the CVM interpreter until they are determined to be hot spots, and then are compiled into native code.

After merging both compilers, we could compile some methods by the AOTC before runtime and compile some methods by the JITC at runtime (which we call

AOTC methods and JITC methods, respectively). Calls between JITC methods and AOTC methods are handled appropriately by executing additional code to meet the calling conventions between them; we do not use the JNI (Java native interface) for interoperation of the translated C code and Java code since JNI would be too slow. The details of the JITC, the AOTC, and the hybrid environment will be described in detail in Secs. 3 and 4.

Based on this hybrid environment, we experimented with a dual-component Java system using conventional Java programs, by compiling the Java system library classes with the AOTC, while handling regular application classes with the JITC. This is assuming that the system classes correspond to the middleware on the client device, while the application classes correspond to downloaded classes from the service provider. We thought that this experimental setup is a reasonable simulation of our target embedded environment since the interaction between system classes and application classes would exhibit a similar behavior to the interaction between middleware classes and downloaded classes (unfortunately, this is not exactly true, as we will see later).

We compare the performance of this hybrid environment (hybrid) with the performances of the full-AOTC and the full-JITC environments, where both library classes and application classes are handled solely by the AOTC and by the JITC, respectively. In this way, we can evaluate the constructed hybrid environment.

Our experimental environment is as follows. The experiments were performed with the AOTC and the JITC implemented on CVM reference implementation (RI). Our CPU is a MIPS-based SoC called AMD Xilleon which is popularly employed in Digital TVs. The MIPS CPU has a clock speed of 300 MHz and has a 16 KB I-cache/16 KB D-cache, with a 128 MB main memory. The OS is an Embedded Linux. The benchmarks are SPECjvm98 (except for javac[a]) and EEMBC.[b]

Figure 1 shows the performance ratio of the full-JITC, the hybrid, and the full-AOTC, compared to the performance of interpreter execution (full-interpreter) as 1.0x for each benchmark. The performance of the full-AOTC, which is an average of 4.0x over the interpreter, is consistently higher than the performance of the full-JITC, which is an average of 2.7x over the interpreter, as we expected. The problem is the performance of the hybrid. Contrary to our expectation, its performance is not positioned in-between of the full-JITC's and the full-AOTC's, and even lower than the full-JITC's in many benchmarks. This is somewhat surprising since we expected a performance at least better than the full-JITC's, by handling library methods using the more powerful AOTC while handling others by the same JITC used in the full-JITC environment.

[a] Javac could not run in our MIPS board in the hybrid mode due to memory overflow caused by one huge method whose JITC requires huge memory.

[b] Any performance numbers for these benchmarks shown in this paper are relative numbers to demonstrate the value of our environment, so they should not be interpreted as official scores.

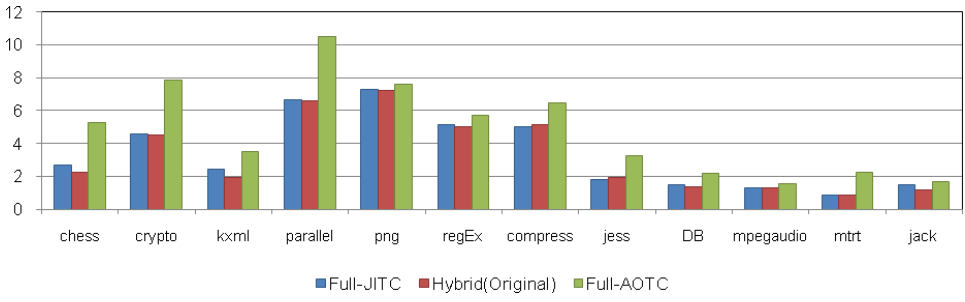

Fig. 1. Performance of the full-JITC, the full-AOTC, and the hybrid with AOTC and JITC, compared to the interpreter (1.0x).

We may speculate on the reasons for this performance anomaly as follows. One possibility would be the call overhead between AOTC methods and JITC (interpreter) methods due to additional code executed for meeting the calling conventions. Or, it might be related to JVM features such as garbage collection (GC) or exception handling (EH), which should work correctly even in the hybrid environment (e.g., an exception raised by an AOTC method can be handled by a JITC method, or GC can occur when the call stack is mixed with AOTC methods and JITC methods). This might cause additional overhead in the hybrid environment for their correct operation. Finally, it might be related to the characteristics of library classes such that their performance would not increase when they compiled with AOTC, compared to when compiled with JITC.

In order to find out the major reason(s) for the performance anomaly of our hybrid environment, we first need to understand how the AOTC, the JITC, the interpreter, and the hybrid environment are constructed. We will describe them in the next two sections, especially focusing on the issues raised above.

3. The JITC and the AOTC

Although our AOTC and JITC target the same CVM, they have been developed independently without any consideration of hybrid execution (in fact, no JITC or AOTC has been developed considering hybrid execution, as far as we know, and it is not clear at this point how to build such a hybrid-execution-aware JITC and AOTC, as will be discussed later). On the other hand, each compiler was developed as reasonably and generally as possible for its best performance benefit, so although the experimental results in Sec. 2 were obtained with our specific implementation, a similar result is likely to be expected with other implementations. This section describes both compilers, especially focusing on their calling conventions, optimizations, and how GC and EH are handled. We start with an overview of the JVM and the CVM interpreter.

3.1. *JVM and the interpreter*

These become local variables of the callee, and are followed by the callee's other local variables, method/frame information of the callee, and the callee's operand stack. When a method returns, the callee's stack frame is popped and the return value is copied from the callee's operand stack to the top of the caller's operand stack.

As a GC-based, object-oriented language, garbage objects are reclaimed automatically. GC requires tracing all reachable objects from the root set to reclaim all unreachable objects. The root set is composed of operand stack slots and local variables of all methods in the call stack (Java stack) and static variables, whose types are object references.

The CVM requires all threads to wait at their GC-point before it performs GC,[10] which is a point in the program where GC can possibly occur. Examples of GC-points include memory allocation requests, method calls, loop backedges, etc. So, a thread should check if there is any pending GC request whenever they pass through a GC-point, and wait there if there is one. When all threads wait at their GC-point, the CVM can start GC by first computing the root set. For this computation, GC needs a data structure describing the location of each root at the GC-point, which is called a GC-map. When GC occurs during interpretation, the interpreter is supposed to analyze the bytecode for each method in the call stack and to compute the GC-map at each GC-point in the method (it saves the GC-maps at the method block for their reuse when GC occurs again). When GC occurs, this GC-map is consulted to decide which stack slots and local variables in the call stack are reference-typed thus being included in the root set. Reference-type static variables are already included in the root set.

Java supports EH such that when an error occurs in a try block, the error is caught and handled by one of subsequent catch blocks associated with the try block. One problem is that the exception throwing try block and the exception-handling catch block might be located in different methods on the call stack, so if no catch block in the method where the exception is thrown can handle the exception, the CVM searches backward through the call stack to find a catch block which can handle it. This mechanism is called stack unwinding[11] and is performed by the exception handler routine included in the CVM interpreter.[10]

3.2. *The JITC*

Our JITC uses adaptive compilation, where a method is initially executed by the CVM interpreter until it is determined as a hot spot method.[9] Then, the method is compiled into native code, which then resides in the memory and is reused whenever the method is called again thereafter. Our JITC performs many traditional optimizations for the compiled method including method inlining. The operand stack slots and the local variables are allocated to registers, with copies corresponding to pushes and pops being coalesced aggressively.

As to the JITC calling convention, all machine registers mapped to the operand stack locations and the local variables at the time of a method call are first spilled to the Java stack (to their mapped locations) before the call is made. Consequently, the Java stack is maintained exactly the same as in the case of interpreter execution during method calls. This is for simplifying argument passing for calls between interpreted methods and JITC methods. Moreover, GC and EH can be handled more easily with this calling convention, as will be explained shortly.

As to GC, unlike in the interpreted methods, there is no GC-time computation of the GC-map in the JITC methods; instead the JITC itself computes and saves a GC-map at each GC-point during translation by checking which Java stack locations (not registers) have a reference at that GC-point. This is so since all registers mapped to the Java stack locations are also spilled to the Java stack at a GC-point if there is a pending GC request (exactly as in method calls; actually, a method call itself is a GC-point). So the GC-map in JITC methods includes only Java stack locations as in the interpreted method and the Java stack is maintained the same when GC occurs.

As to EH, the exception handling routine in the CVM interpreter is supposed to handle exceptions even for JITC methods such that if an exception occurs in a JITC method, it will jump to the handling routine which will perform stack unwinding. When a catch block is found in a JITC method, the bytecode of the catch block will be executed by the interpreter, so there is no need to compile the catch block by the JITC. This is fine since an exception would be an "exceptional" event, so the performance advantage of executing compiled catch blocks would be little. This interpreter-based execution of catch blocks requires the Java stack to be maintained exactly the same as in the interpreter execution, so registers are also spilled to the Java stack when an exception occurs even in a JITC method.

3.3. *The AOTC*

Our AOTC translates the bytecode of classes into C code, all of which are then compiled and linked together with the CVM source code using gcc to generate a new CVM executable.[8] We took this particular approach of AOTC rather than other alternatives considering a few aspects, as explained below.

We took the approach of bytecode-to-C[5] rather than bytecode-to-native,[4] since we can resort to an existing compiler for native code generation, which allows a faster time-to-market and a better portability. In addition, we can generate high-quality code by using full optimizations of gcc, which would be more reliable and powerful than our own optimizer. In fact, most AOTCs take the approach of bytecode-to-C,[2,3,12,13] including commercial ones such as Jamaica,[14] IBM WebSphere Real-time VM,[15] PERC,[16] and Fiji,[17] so we believe that anyone who wants to build a hybrid environment is likely to employ the b-to-C AOTC. Our AOTC also performs some Java-specific optimizations that gcc cannot handle, such as elimination of redundant null pointer checks or array bound checks.[8]

We statically compile and link every translated C code with the CVM source code, instead of compiling each C code separately and loading its machine code to the CVM dynamically at runtime. This allows method calls or field accesses to different classes to be resolved at translation time, obviating runtime resolution. Also, inlining between different classes is much easier.

In our AOTC, each local variable and operand stack slot is translated into a C variable with a type name attached. For example, a reference-type operand stack slot 0 is translated to s0_ref. We then translate each bytecode to a corresponding C statement, while keeping track of the operand stack pointer. For example, aload_1 which pushes a reference-type local variable 1 onto the stack is translated into a C statement s0_ref = l1_ref; if the current stack pointer is zero when this bytecode is translated.

The calling convention of our AOTC follows the format of a regular C function call. That is, a method call in the bytecode is translated into a C function call whose name is composed of the Java class name and the method name (similar to a JNI method naming convention). The argument list consists of an environment variable for capturing the CVM state, followed by regular C variables corresponding to the argument stack locations at the time of the call. Such a C function call will be compiled and optimized by gcc and arguments will be passed via registers or the C stack, so AOTC calls will be much faster than JITC calls or interpreter calls. Our AOTC also performs inlining for some method calls.

As to GC, since our AOTC translates stack slots or local variables that have root references into C variables, it is difficult to know where gcc will place those variables in the final machine code. So the AOTC cannot make a GC-map. Our solution is generating additional C code that saves references in the Java stack frame whenever a reference-type C variable is updated such that when GC occurs, all Java stack slots of AOTC methods constitute a root set.[6] For this purpose, a Java stack frame is still allocated and extended during the execution of AOTC methods, although the machine code of AOTC methods (including calls) is based only on the C stack and the registers. In order to reduce the runtime overhead caused by the additional C code, we perform optimizations to reduce the number of references saved in the stack frame and the number of stack extensions.

There is one more issue in GC with the AOTC. Since the CVM employs a moving GC algorithm, objects can be moved during GC. CVM GC is supposed to update the addresses of moved objects for those references saved in the Java stack frame, but not the updated reference C variables. So after GC, we need to copy the addresses from the Java stack frame back to the reference C variables, which require additional C statements.

As to EH, when an exception occurs in an AOTC method, the environment variable will be set appropriately and the control will transfer to a catch block if the method has one that can handle it. If there is no catch block, the method simply returns to the caller. In the caller, we check if an exception occurred in the callee and

if so, we try to find an appropriate catch block in the caller. If there is no catch block, then the method also returns and this process repeats until a catch block is found. This means that we need to add an exception check code right after every method call, which would certainly be an overhead. However, it is a simple check and merged with the GC check for copying references back, so the overhead is not serious.[7]

4. Hybrid Compilation Environment

Previous section described the AOTC and the JITC, each of which has been developed efficiently for its own performance advantage. This section describes how we merged them to build a hybrid environment. We address how method calls, GC, and EH are made across different environments. Since the interpreter and the JITC have already worked collaboratively together for adaptive compilation, the merge across AOTC and JITC (interpreter) would be the primary issue of a hybrid environment.

When there is a call between a JITC method and an AOTC method, some reconciliation process is needed to meet the calling convention between them, especially for parameter/return value passing and the stack management. Since an AOTC method passes parameters using registers and the C-stack, while a JITC method passes parameters using the Java stack, appropriate conversion between them is required.

When a JITC method calls an AOTC method, a stack frame for the AOTC method is first pushed on the Java stack and is marked as an AOTC frame. Then, parameters in the Java stack of the JITC method are copied into registers (and into the C-stack if there are more than four parameters in MIPS). Finally, a jump is made to the AOTC method entry. This process is depicted in Fig. 2 and is performed by an assembly function. When an AOTC method returns back to a JITC method, a similar process is needed to copy the return value in a register of the AOTC method to the Java stack of the JITC method.

If there are updates of reference C variables in the AOTC method, the Java stack is extended to accommodate those references for preparation of GC, as explained in Sec. 3.3 (depicted as a reference stack in Fig. 2). If the AOTC method calls another AOTC method which also updates reference C variables, the Java stack frame is extended again. Consequently, a single stack frame is shared among consecutively called AOTC methods and is extended for saving references whenever necessary.

When an AOTC method calls a JITC method, a new stack frame is pushed on the Java stack (marked as a JITC frame), and the parameters that are in C variables are copied to the operand stack and a jump is made to the JITC method.

When GC occurs in a hybrid environment, all the root references are guaranteed to be located in the Java stack, but identifying Java stack slots that contain references depends on the type of the stack frame. If it is an AOTC frame, all stack locations will have references since we saved only references there. If it is a JITC method or an interpreter method, we check with the GC-map at the GC-point to tell the stack slots which have references.

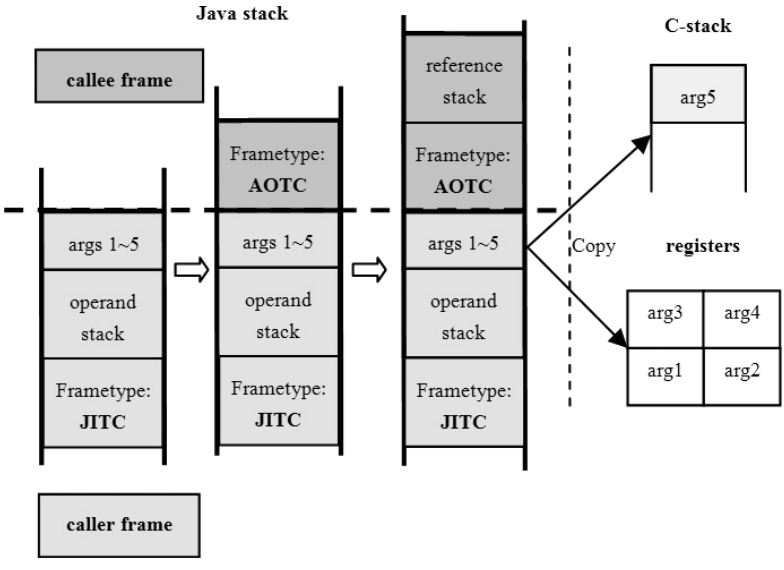

Fig. 2. AOTC method call from interpreter in MIPS.

Handling exception across AOTC and JITC methods is relatively simple, which is done with exception checks for both AOTC and JITC methods. When a JITC method calls an AOTC method, there should be an exception check added right after the call, as we did in an AOTC method. If an exception occurs somewhere in the call chain at an AOTC method and if it is not caught before returning to this JITC method, it will be checked at the added exception check code. Then, the control is transferred to the exception handling routine of the interpreter, which performs stack unwinding starting from this JITC method.

When an AOTC method calls a JITC method, we add a check code right after the call as usual. If an exception occurs somewhere in the call chain at a JITC method, the exception handling routine of the interpreter will perform stack unwinding. It checks for each method on the call stack, one by one, if there is a catch block who can handle the exception. It can also tell if a method on the call stack is a JITC method or an AOTC method using the frame type, so when the caller AOTC method is eventually met during stack unwinding, the interpreter will simply make a call return, which will transfer the control back to the AOTC method as if the JITC method returns. Then, the exception check code is executed and the normal AOTC exception handling mechanism based on the exception check proceeds.

As one can notice easily, supporting EH or GC correctly in the hybrid environment causes a relatively little overhead since they are essentially the same as in AOTC and JITC. On the other hand, method calls may cause some overhead due to different calling conventions. We will analyze method calls in the following section.

5. Analysis of the Hybrid Environment

Previous section described our hybrid environment, which we think is a reasonable merge of high-performance JITC and AOTC. In this section, we analyze our performance results in Sec. 2 to understand the root causes of its performance anomaly.

5.1. *Call behavior of benchmarks*

We first examine the call behavior of our benchmarks. Figure 3 depicts the distribution of calls and execution time between application methods and library methods in each benchmark. The call distribution varies widely from benchmark to benchmark. Since most calls in crypto, png, regEx, compress, jess, mpegaudio, and mtrt are application method calls, compiling library methods by AOTC in the hybrid would not improve performance much compared to the full-JITC. However, most calls in parallel, DB, and jack and around half of the calls in chess and kxml are library calls, although the library execution time takes a less portion. So we should expect some reasonable performance improvement at least for these benchmarks with the hybrid compared to the full-JITC. Unfortunately, the hybrid led to worse performance in these benchmarks as well as the first set of benchmarks, as we saw previously in Fig. 1.

Generally, our benchmarks tend to spend more time in applications rather than in libraries, which would not exactly be the case in the middleware and downloaded classes. As the middleware gets more substantial, much of the execution time should be spent in the middleware, as explained in Sec. 2. However, even when the libraries are dominant as in db and jack, Fig. 1 shows that the hybrid performs worse than the full-JITC. In fact, even when the libraries are not dominant, there should be, in theory, no performance degradation compared to the full-JITC. We suspect the call overhead as one reason, as analyzed below.

5.2. *Call overhead*

In the hybrid environment, we classified the call types. We measured how many calls are from JITC methods to AOTC methods, from AOTC to AOTC methods, from

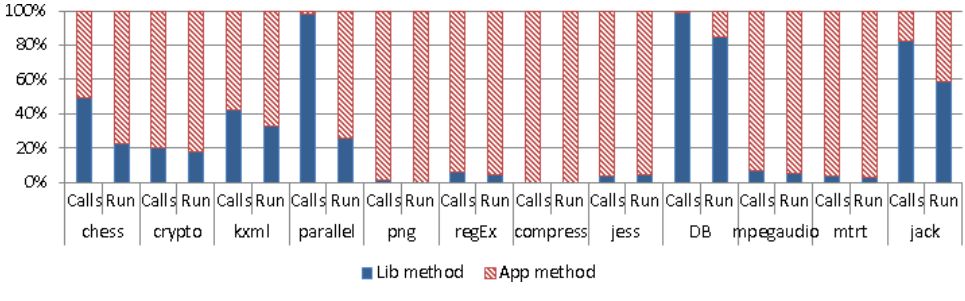

Fig. 3. Call behavior of each benchmark.

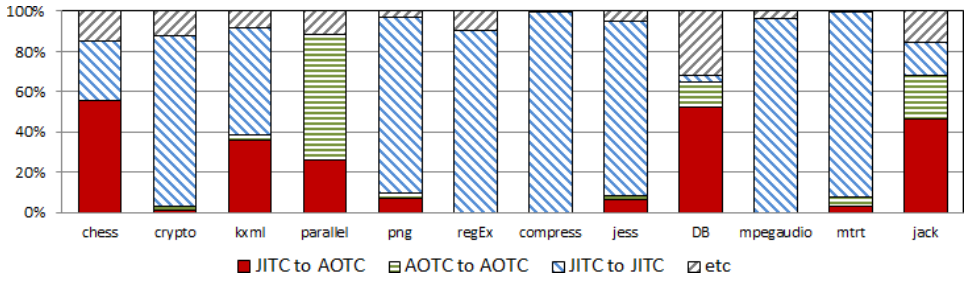

Fig. 4. Call count distribution among different types of calls.

JITC to JITC methods, and others (e.g., JNI methods calls) which is shown in Fig. 4. Those benchmarks which include many library calls have many JITC-to-AOTC calls, as expected.

We then measured the call overhead from a JITC method to an AOTC method (J-to-A), which is supposed to occur frequently in our hybrid environment, compared to the call overhead of a JITC method to a JITC method (J-to-J). For this evaluation we made a simple Java method which makes a return. We compiled this method with our AOTC and made a JITC method to call it five million times with a variable number of arguments. We measured the running time and then isolated the loop and argument pushing overhead in order to identify the J-to-A call overhead only. Then, we compiled this method with the JITC and measured the J-to-J call overhead similarly. Finally, we compiled all methods with the AOTC and measured the A-to-A call overhead. We experimented both with a static method call and a virtual method call.

Figure 5 depicts the call overhead (in microseconds) of a single J-to-A call, J-to-J call, and A-to-A call for the static method call and the instance method call. It shows that the call overhead of J-to-A is around 2.5 times to that of J-to-J. It is also shown that the J-to-A call overhead increases as the number of arguments increases, due to argument copying. We also estimated the total J-to-A call overhead during

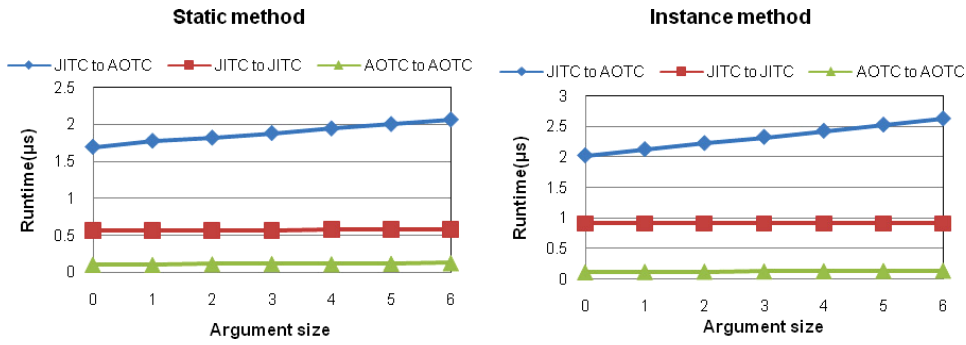

Fig. 5. The call overhead of invokestatic and invokevirtual.

execution, by multiplying a single J-to-A call overhead and the J-to-A call counts. Its ratio to the whole running time is found to be significant for chess (20%), kxml (23%), db (19%), jack (16%), and parallel (7%) which have many J-to-A calls.

These results indicate that J-to-A calls are much slower than J-to-J calls and they take a significant portion of running time when they are frequent. We estimated the execution time of some hypothetical "faster" hybrid environment when the J-to-A call overhead is replaced by the J-to-J call overhead (i.e., for each J-to-A call, compute the overhead difference from a J-to-J call with the same number of arguments, multiply it by the call count, and subtract the result from the execution time of the original hybrid environment). Figure 6 shows the performance of such a faster hybrid environment, compared to those of the original hybrid and the full-JITC. The faster hybrid outperforms the original hybrid tangibly in chess, kxml, parallel, DB, and jack, which have a higher ratio of the total J-to-A call overhead to the running time, and even outperforms the full-JITC in parallel and DB. Consequently, it appears that the J-to-A call overhead significantly contributes to the performance degradation of the original hybrid.

The J-to-A call overhead is primarily due to interfacing the AOTC calling convention based on the C stack and the JITC calling convention based on the Java stack. It might be argued that this interface problem would not occur if we perform AOTC using the JITC module, with the full-fledged optimizations that are missing in JITC enabled. This JITC-based AOTC would certainly obviate the interface problem for J-to-A calls, yet there is one important issue in our hybrid environment. That is, even for every A-to-A call, the caller first needs to spill registers to the Java stack so as to keep the Java stack exactly the same as in interpreter execution, as we did with JITC methods. This will certainly slow down A-to-A calls, which are supposed to occur frequently with a substantial Java middleware, and would affect the performance negatively. Another minor issue is that we cannot use the approach of bytecode-to-C anymore, losing many of its advantages described in Sec. 3.3, such as faster time-to-market, portability, and powerful and reliable optimizations with an existing compiler.

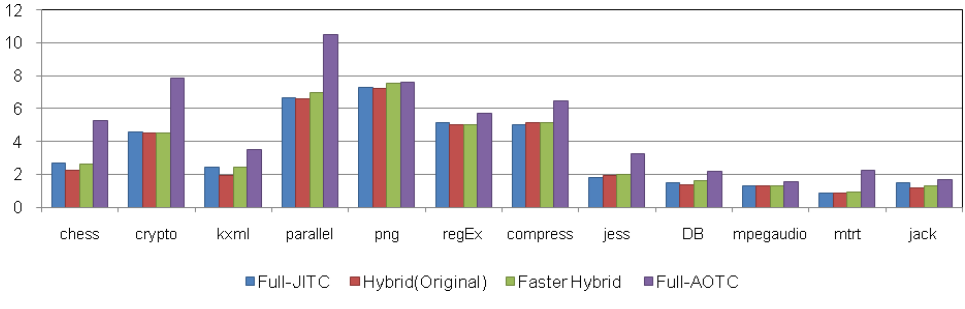

Fig. 6. Performance of hybrid environment, faster hybrid environment, and full-JITC.

5.3. *Application methods and library methods*

We also tried with an opposite hybrid environment where we compiled application methods by the AOTC and handled library methods by the JITC. This is for understanding the characteristics of library methods and application methods in terms of their profitability achievable by AOTC compared to by JITC.

Figure 7 shows the performance of a new hybrid environment compared to the full-JITC and the full-AOTC. This time this hybrid environment exhibits a performance level in-between of the full-JITC's and the full-AOTC's for crypto, parallel, compress, jess, mpegaudio, and mtrt, where application methods are dominant (application method calls in parallel are scarce, yet the execution time of application methods takes more than 70%, as seen in Fig. 3). For other benchmarks where application methods are not dominant, the hybrid performance is still lower than the full-JITC's. This also appears to be due to the call overhead from AOTC methods to JITC methods, which is even higher than JITC-to-AOTC calls.

It is questioned why the hybrid environment can improve the performance of the full-JITC when compiling applications by AOTC for application-dominant benchmarks, while it cannot when compiling libraries by AOTC for library-dominant benchmarks. One possible reason is that hot application methods often include computation loops, which will be better optimized when compiled by AOTC than by JITC, since AOTC includes more powerful optimizations. On the other hand, hot library methods tend to have no computation loops but are called many times, so the benefit of compiling them by AOTC is easily offset by the J-to-A call overhead. For example, the hot library methods in db are called frequently but they either have no loops or have a loop which calls many methods, so the benefit of AOTC for them would be small.

Consequently, the characteristics of methods are also important in deciding whether their performance could be improved when compiled with AOTC compared to when compiled with JITC. This can be used effectively in choosing AOTC candidates. In fact, the real characteristics of the middleware and downloaded classes

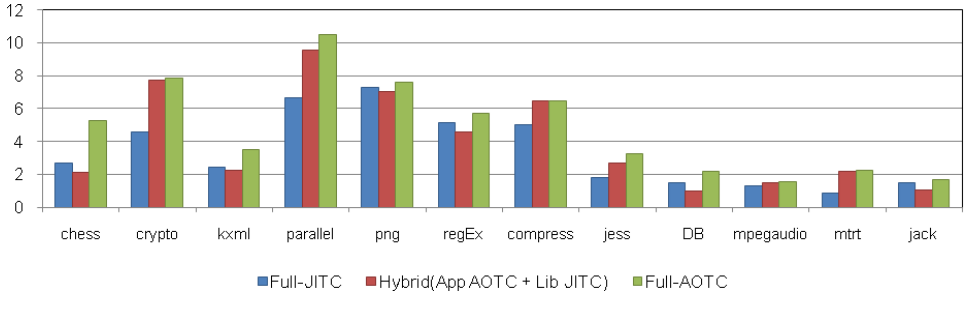

Fig. 7. Performance of another hybrid environment that AOTC applications and JITC libraries.

might be different from those of the library and the application used in our experiment, so we need to investigate the real cases further.

5.4. *Improving hybrid performance*

Previous section analyzed the performance problems of the hybrid environment. In this section, we investigate how we could possibly achieve the desired hybrid performance. We first describe how to reduce the JITC-to-AOTC call overhead. We then explore the performance impact of the distribution of JITC methods and AOTC methods.

5.4.1. *Reducing the JITC-to-AOTC call overhead*

The analysis result in the previous section indicates that the call from JITC methods to AOTC methods is problematic since its overhead is higher than other type of calls. One solution would be reducing the JITC-to-AOTC call overhead itself. The overhead of copying arguments from the operand stack of the JITC method to the registers and the C stack of the AOTC method by an assembly routine appears to be substantial. We can reduce this overhead by allowing the callee to access the caller's operand stack directly for retrieving the argument. This can increase the overhead of AOTC-to-AOTC calls slightly, though, since an AOTC method should first check if its caller is an AOTC method or a JITC method. In fact, our AOTC is designed to maximize its performance only, including the AOTC-to-AOTC calls, so it would be reasonable to slow down AOTC-to-AOTC calls slightly to increase the overall performance of a hybrid environment.

We actually implemented this idea and experimented with it. Figure 8 shows the performance of the new calling convention, hybrid(Mix), compared to the original hybrid, hybrid(Original), the full-JITC, and the full-AOTC. For those benchmarks whose JITC-to-AOTC calls are frequent such as chess, kxml, DB, and jack (where there were improvements in Fig. 6), the hybrid(Mix) improves the performance tangibly, outperforming the full-JITC in some benchmarks. Comparing this graph

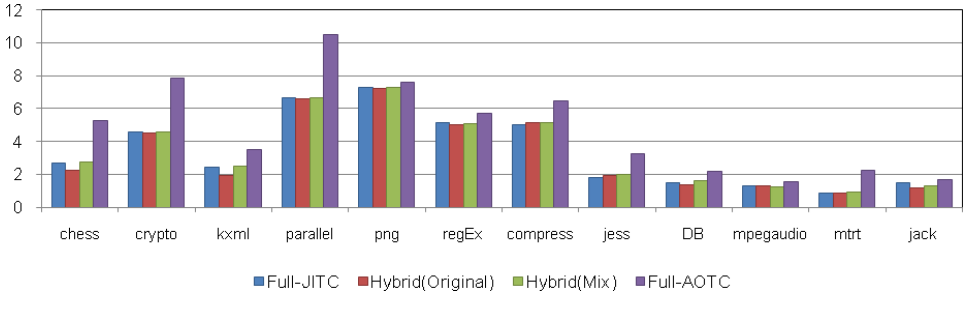

Fig. 8. Performance of mixed argument passing in AOTC.

with Fig. 1, we can see that most of the anomaly results are gone away, placing the hybrid performance at least equal to the full-JITC performance in most benchmarks. However, we still could not see any hybrid performance that is definitely better than the full-JITC's.

5.4.2. *Performance impact of the distribution of JITC methods and AOTC methods*

Although the call overhead optimization in the previous section could remove the performance degradation of the hybrid environment, placing its performance closer to the full-AOTC's would be dependent upon the distribution of execution time among JITC methods and AOTC methods. That is, if we spend more time in the AOTC methods than in the JITC methods, the benefit of AOTC over JITC will take effect, leading to a higher performance than the full-JITC's. In embedded Java platforms, this means that the middleware is well-designed such that downloaded classes are implemented mainly by calls to the middleware rather than by their own computations, which allows spending more time in the middleware than in the downloaded classes. In this section, we want to explore the impact of the distribution of JITC methods and AOTC methods on the performance of the hybrid environment.

For this experiment, we compiled the library methods by the AOTC as previously. Then, we compile additional application methods by the AOTC, depending on their call depths from the main method. That is, we measure the minimum call depth of each method (e.g., if a method has a call depth of three for a call chain and four for a different call chain, its minimum call depth is three), and if it is higher than a given threshold T, we compile it by the AOTC. The remaining application methods will be compiled by the JITC as usual. Consequently, a lower T value will make more methods to be compiled by the AOTC. Since we have the estimated execution time profile of each method, we can sum up the distribution of JITC methods and AOTC methods, and we can understand the relationship between the hybrid performance and the distribution.

Figure 9 shows the hybrid performance of the EEMBC benchmark with a diverse T value. We experimented with $T = 2, 4, 8, 12$, and 16 (we use the calling convention of Sec. 6.1). When $T = 8$, for example, we perform AOTC for those application methods whose minimum call depth is higher than or equal to 8, in addition to the library methods. For each T value, each graph also includes the proportion of the estimated execution time of AOTC methods to the estimated execution time of all (AOTC + JITC) methods. As T decreases, more methods are compiled by the AOTC (the proportion of AOTC methods comes closer to 1), boosting the hybrid performance closer to the AOTC performance (we could observe a similar results for the SPECjvm98 benchmarks). These graphs indicate that the distribution of AOTC methods and JITC methods affect our hybrid performance seriously, meaning that the hybrid compilation can be effective only when enough running time is spent in the middleware, which are compiled by the AOTC.

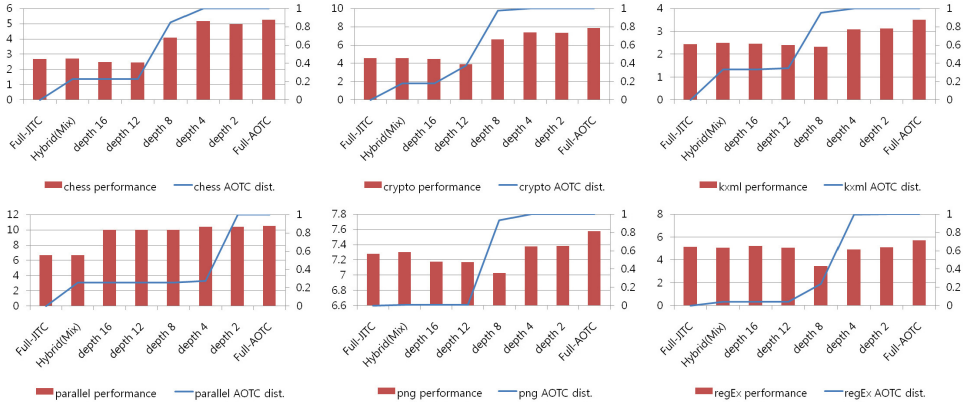

Fig. 9. Performance of hybrid compilation with different T values for the EEMBC benchmarks.

We actually studied the DTV environment of Korea using the data broadcasting of the MBC channel. There are four menus in its data broadcasting 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. Our analysis shows that more than 90% of methods executed belong to the system classes or the ACAP middleware classes, and xlet methods take a small portion.[19] Although we could not measure the exact execution time, we can be assured that much of the running time is spent in the system classes and the middleware classes, which can be compiled by the AOTC. This distribution behavior might change in the future as more substantial xlets for sophisticated data broadcasting (e.g., T-commerce) are introduced, but it is likely that enough running time will continue to be spent in the AOTC methods, justifying the adoption of the hybrid compilation environment.

6. Comparison to Related Work

We can say that existing JITCs and AOTCs already employ some form of a hybrid execution environment. For example, most JITCs are using adaptive compilation where the interpreter is used for hot spot detection. In addition, most AOTCs require the interpreter for supporting dynamic class loading. However, this interpreter execution is simply for supplementing JITC or AOTC, so this is not a genuine form of a hybrid environment.

QuickSilver is a quasi-static compiler developed for the IBM's Jalapeno system for servers.[18] It saves all JITC methods in the files at the end of execution, and loads them directly without JITC when they are used in later execution. Therefore, it employs a form of AOTC, yet it is based on a JITC, not on a separate offline compiler. There are no interface issues between JITC methods and AOTC methods in this hybrid environment because their machine code is generated by the same

compiler, unlike our AOTC−JITC hybrid environment. However, the benefit is merely reducing the JITC overhead without any improvement for the AOTCed code quality. Actually, there still is the class loading overhead of constant pool (CP) resolution or building class data structures for the AOTCed classes, unlike our bytecode-to-C AOTC where the translated C code is compiled together with the JVM source, hence no such overhead (already-resolved CP entries and class data structures are romized in the JVM). This also obviates any loading process of the AOTC machine code into the memory, while QuickSilver can suffer from a relocation and CP resolution overhead during the loading process.

There is a commercial JVM that takes a similar approach to QuickSilver. Sun's phoneME Advanced has an AOT option which allows compiling a list of prechosen methods using its JITC module and saves their machine code in a file on a persistent storage. When the JVM starts officially, it will use the compiled machine code directly without interpretation or JITC, when they are executed. There are no interface issues between AOT methods and JITC methods since they are based on the same compiler as in QuickSilver. However, unlike QuickSilver, the machine code is generated statically irrespective of program execution, so the AOT-generated code is not exactly the same as the JITC-generated code but worse. For example, the AOT inlining is inefficient since it is not based on the runtime profile information unlike JITC. Moreover, a few code optimizations in JITC are disabled due to the relocation and code patch issues. Fundamentally, JITC would not perform any time-consuming optimizations since the compilation overhead is part of the running time, so a JITC-based AOTC is likely to underperform the bytecode-to-C AOTC, as indicated by the comparison graph of the full-AOTC and the full-JITC in Fig. 1.

We actually constructed a hybrid environment using the AOT for a commercial DTV software platform where the downloaded xlets are handled by the JITC, while the ACAP middleware and system classes are compiled by the AOT.[19] We observed little performance improvement compared to the full-JITC's for commercial xlets broadcasted in Korea because of the problems listed above, although the AOT obviates the JITC overhead for the AOT methods.

Jikes RVM includes two kinds of compilers: a baseline compiler and a tiered set of optimizing compilers.[20] The baseline compiler translates bytecode into machine code before execution starts, while the optimizing compilers recompile hot methods with optimizations at runtime. So, the baseline compiler and the optimizing compiler correspond to an AOTC and a JITC, respectively. However, what the baseline compiler generates is machine code corresponding to what the interpreter does, so the relationship between the two compilers is more like our JITC-interpreter, not our JITC−AOTC.

The .NET platform of Common Language Runtime VM also employs a JITC, which translates MSIL (MS intermediate language) into machine code.[22] It is also possible to invoke the JITC offline so as to compile ahead-of-time. This JITC-based AOTC can save only the JITC overhead, as QuickSilver can.

7. Summary and Future Work

We believe that an embedded Java platform architecture with the middleware classes on the client device and downloaded classes from the service provider will be a mainstream. This paper proposes a hybrid compilation environment with both AOTC and JITC for accelerating this dual-component software architecture. As far as we know, our work is the first one which proposes a hybrid compilation environment, after identifying the trend of the embedded Java software architecture.

We performed a case study by merging an existing JITC and AOTC, yet found some performance anomaly with such a hybrid environment. Our analysis shows that the anomaly is primarily due to method calls between JITC methods and AOTC methods which cause serious call overhead. We also found that it is related to the characteristics of methods. An optimization to reduce the call overhead could reduce the anomaly, but the desired hybrid performance with our environment appears to be achievable only when enough running time is spent on the AOTC methods. Fortunately, the middleware and system classes are dominantly executed in a DTV environment, justifying the proposed hybrid compilation.

Fundamentally, it is questioned how we should design an efficient hybrid environment in general. One idea is that we employ a common IR (intermediate representation) for both the JITC and AOTC so as to reduce the interface issues between them. However, it appears that one root problem of the hybrid environment is still related to the Java stack that we should maintain around the method call, GC, and EH boundaries. We can think of three cases.

If all of our compilers are based on the Java stack, meaning that the AOTC is based on the JITC as in Sun's AOT, yet with full-fledged optimizations added for the purpose of the AOTC, there will be no interface problem and a better performance will be achievable than a JITC-only environment. However, the method call with the Java stack will be slower than the one with the C stack with register-based argument passing (as in our AOTC-to-AOTC calls), unable to achieve the best performance.

If the AOTC is based on the C stack while the JITC is based on the Java stack as in our proposed environment, there is a serious interface problem as discussed in this paper.

If all of our compilers are based on the C stack, even with the interpreter being replaced by a low-end JITC, then we can achieve a high performance with register-based argument passing and with no interfacing problems. The question is if we can redesign GC and EH easily so that they work correctly and efficiently even without the Java stack. Also, it is questioned if such a JITC-only-based AOTC can always achieve a fast call, as the AOTC-to-AOTC call in our bytecode-to-C AOTC, considering that its AOTC code should also work with the low-end, JITC-generated code; this might lead to slower JITC-to-AOTC calls than its AOTC-to-AOTC calls probably due to the additional overhead of copying arguments to registers similarly as in our hybrid environment, unless the low-end JITC can perform

high-performance register allocation. Finally, the class loading overhead and the machine code loading overhead discussed in the QuickSilver case might affect the AOTC performance negatively compared to the bytecode-to-C. Consequently, this stack issue should be resolved first for building an efficient hybrid environment.

Another way of building a compilation environment based only on the C-stack is performing AOTC for the downloaded classes at the provisioning server and sending the compiled binary to the client. In fact, there is a commercial Java platform for mobile phones called WIPI (Wireless Mobile Platform for Interoperability) 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. Obviously, there is no stack interface issue between downloaded classes and middleware classes, but downloading binaries instead of Java classes can raise portability and bandwidth issues, thus being rarely used.

Another idea is performing a C-stack-based AOTC for the downloaded classes at the client. In fact, we performed AOTC for the downloaded xlets for the DTV during the idle time of the DTV,[19] yet its AOTC was based on the Java-stack-based JITC. If we employ a separate, C-stack-based AOTC for the downloaded classes and if there is indeed an idle time, we can have a C-stack-based binaries for both the middleware and downloaded classes. One issue is that since the AOTC is performed on the online client device, not on an offline server, such an AOTC would have both time and memory constraints, as the low-end JITC mentioned above. So it might have similar performance issues.

As to the code optimization, JITC has the advantage of fully exploiting the runtime profile information with its dynamic compilation, while the bytecode-to-C can perform profile-based static compilation only, so the JITC-based AOTC might lead to better code quality. However, this can be exploited only when we obtain the machine code after the execution of programs as in QuickSilver, not in Sun's AOT, thus requiring the class loading overhead and the machine code loading overhead as well. Actually, no matter which AOTC approach is taken, we need more elaborate code optimizations to improve the hybrid performance, such as the collaborative, static, and dynamic optimizations between AOTC and JITC, similar to but extending the idea of dynamic compilation.[21] Investigation of these issues is also left as a future work.

Acknowledgments

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0026481).

References

1. J. Aycock, A brief history of just-in-time, *ACM Comput. Surv.* **35** (2003) 97–113.
2. 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, *Proc. USENIX Conf. Object-Oriented Technologies and Systems*, Portland, Oregon (1997), p. 3.
3. G. Muller, B. Moura, F. Bellard and C. Consel, Harissa: A flexible and efficient Java environment mixing bytecode and compiled code, *Proc. USENIX Conf. Object-Oriented Technologies and Systems*, Portland, Oregon (1997), p. 1.
4. M. Weiss, F. Ferrière, B. Delsart, C. Fabre and F. Hirsch, Turbo J, a Java bytecode-to-native compiler, *Proc. ACM SIGPLAN Work. on Languages, Compilers, and Tools for Embedded Systems* (1998), pp. 119–130.
5. A. Varma and S. S. Bhattacharyya, Java-through-C compilation: An enabling technology for Java in embedded systems, *Proc. Conf. Design, Automation and Test in Europe* (2004), p. 30161.
6. D. Jung, S. Bae, J. Lee, S. Moon and J. Park, Supporting precise garbage collection in Java bytecode-to-C ahead-of-time compiler for embedded systems, *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Korea, Seoul (2006), pp. 35–42.
7. D. Jung, J. Park, S. Bae, J. Lee and S. Moon, Efficient exception handling in Java bytecode-to-C ahead-of-time compiler for embedded systems, *Comput. Lang. Syst. Struct.* **34** (2008) 170–183.
8. D. Jung, S. Moon and S. Bae, Design and optimization of a Java ahead-of-time compiler for embedded systems, *Proc. Int. Conf. Embedded and Ubiquitous Computing*, China, Shanghai (2008), pp. 169–175.
9. S. Lee, S. Moon and S. Kim, Enhanced hot spot detection heuristics for embedded Java just-in-time compilers, *Proc. ACM SIGPLAN/SIGBED 2008 Conf. Languages, Compilers, and Tools for Embedded Systems*, USA, Tucscon (2008), pp. 13–22.
10. Sun Microsystems, Porting guide — Connected device configuration and foundation profile, version 1.0.1 Java 2 platform micro edition (2002).
11. T. Ogasawara, H. Komatsu and T. Nakatani, A study of exception handling and its dynamic optimization in Java, *Proc. ACM SIGPLAN Conf. Object Oriented Programming, Systems, Languages, and Applications*, USA, Tampa Bay (2001), pp. 83–95.
12. A. Armbuster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka and J. Vitek, Real-time Java virtual machine with applications in avionics, *ACM Trans. Embed. Comput. Syst.* **7** (2007) 1–49.
13. A. Nilsson and S. Robertz, On real-time performance of ahead-of-time compiled Java, *Proc. IEEE Int. Symp. Object-Oriented Real-Time Distributed Computing* (2005), pp. 372–381.
14. F. Siebert, Eliminating external fragmentation in a non-moving garbage collector for Java, *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, USA, San Jose (2000), pp. 9–17.
15. M. Fulton and M. Stoodley, Compilation techniques for real-time Java programs, *Proc. Int. Symp. Code Generation and Optimization* (2007), pp. 222–231.
16. T. Schoofs, E. Jenn, S. Leriche, K. Nilsen, L. Gauthier and M. Richard-Foy, Use of PERC Pico in the AIDA avionics platform, *Proc. Int. Work. Java Technologies for Real-Time and Embedded Systems*, Spain, Madrid (2009), pp. 169–178.
17. F. Pizlo, L. Ziarek, E. Blanton, P. Maj and J. Vitek, High-level programming of embedded hard real-time devices, *Proc. European Conf. Computer Systems*, France, Paris (2010), pp. 69–82.

18. M. Serrano, R. Bordawekar, S. Midkiff and M. Gupta, Quicksilver: A quasi-static compiler for Java, *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications* (2000), pp. 66−82.
19. D. Jung, S. Moon and H. Oh, Hybrid Java compilation and optimization for digital TV software platform, *Proc. Int. Symp. Code Generation and Optimization*, Canada, Toronto (2010), pp. 73−81.
20. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo and V. Sarkar, The Jikes Research Virtual Machine project: Building an open-source research community, *IBM Syst. J.* **44** (2005) 399−417.
21. UW Dynamic Compilation Project, http://www.cs.washington.edu/research/projects/unisw/DynComp/www/.
22. R. Wilkes, http://msdn.microsoft.com/msdnmag/issues/05/04/NGen.