모바일 환경을 위한 Java-to-C 컴파일러 구조

한영선° 박인호 황석중 김선욱

고려대학교 전자컴퓨터공학과

고급 컴퓨터 시스템 및 컴파일러 연구실

{youngsun°, htilil, nzthing, seon}@korea.ac.kr

Youngsun Han°. Inho Park. Seokjoong Hwang. Seon Wook Kim

The Structure of Java-to-C Compiler for Mobile Computing Environment

Advanced Computer Systems and Compiler Laboratory
Dept. of Electronics and Computer Engineering, Korea University

Abstract

Java's performance is sometimes not acceptable due to interpretation overhead by the Java Virtual Machine (JVM). In this paper, we present a compiler structure of Java-to-C translator for high performance on resource limited environment like mobile devices. Our compiler framework translates Java into C codes to preserve Java's programming semantics such as inheritance, method overloading, virtual method invocation, and so on. Also our compiler fully supports Connected Limited Device Configuration (CLDC) 1.0 API's. We show that our compiler improves the speedup by up to eleven times more than JVM-only execution in measured benchmarks.

1. Introduction

Despite of the distinguished advantages over other programming languages, there are two shortcomings to use Java, i.e, the size of Java virtual machines and performance limitation due to interpretation. In order to alleviate the performance problem, many methods have been proposed, and such as just-in-time (JIT) compiler and ahead-of-time compiler.

In this paper, we introduce a Java-to-C compiler for high performance on resource limited environment like mobile platforms[1,2]. There are two closely related works in area of Java-to-C translation[3,4]. Toba[3] is a system to generate standalone Java application which were targeted for JDK 1.1. It has a bytecode-to-C translator and additional runtime libraries to support garbage collection, thread management and Java API. In [4], the Java-through-C compilation system for embedded systems has been developed. There are the following differences between our Java-to-C compiler and the others.

During execution of Java application, the mobile system will make serious efforts to dynamically allocate memory for Java object and array structures. When we try to allocate an multi-dimensional array formatted as 10 rows and 10 columns of integer data. While an 100-sized memory block of integer type will be allocated at one time in C program by linearization, several steps will required for Java program. To resolve the problem, our Java-to-C compiler performs profiling to find if the memory block can be allocated statically. If the size of the block can be determined during profiling at program analysis, it will allocate a memory block for a Java object or an array structures statically.

The exception handling is a helpful function to develop Java application. It makes easy to correct errors that were generated at runtime. Some bytecodes are specified in Java language specification to throw exceptions when a certain conditions are satisfied. But, the function is not always necessary. If we assume that a Java application will never throw any exceptions during execution, we will know that the exception handling is unnecessary overhead. Our flexible code generation framework generates different codes according to the execution environment.

2. Structure of Java-to-C Compiler

2.1 Overall Architecture

Our Java-to-C compiler is organized into four components: a class file reader, a bytecode-to-C translator, an application manager, runtime libraries. Figure 1 represents structure of the compiler.

2.2 Classfile Reader

A java compiler compiles a Java file into several class files. After each class file is read into a classfile reader, it is translated into class blocks, which will be used during Java-to-C compilation process. During the translation of a classfile, all its associated classfiles would be translated together. The class block is optimized to maintain information such as fields, methods and super class. certainly needed for the compilation steps.

2.3 Bytecode-to-C Translator

The compilation process of a bytecode-to-C translator is divided into two phases, a preprocessor and a translator. And the translation is performed for each method. The preprocessor profiles the whole bytecode sequence in a method to build up a control flow graph (CFG). The control flow graph is used to adjust status of a virtual stack across its parent and child basic blocks. The stack has several slots to store types of temporary variables and their information is used for naming variable by numbering scheme. In the second phase, each basic block of the graph is translated into C code.

Figure 2 represents the translation process. The prologue maintains a structure of a symbol table for its enclosing method, and the symbol table is initialized properly. Similarly the epilogue finalizes the code emission of the enclosing method. The compiler visits each basic block node, and prints out the translated C codes.

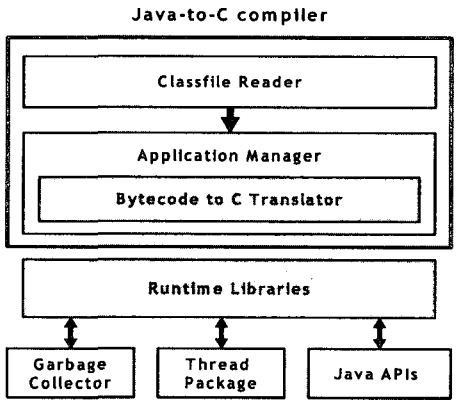

Fig.1. Structure of Java-to-C compiler

2.4 Application Manager

The application manager generates a complete C program using the result from the components previously explained. The generated C program maintains the full features of the Java platform, such as inheritance, method overloading, virtual method invocation, and so on. For code generation, it has prototypes for Java runtime data structures, class initialization, method invocation, garbage collection and a main method to start up C programs. While the code generation is performed, the prototypes are copied into the generated C programs properly.

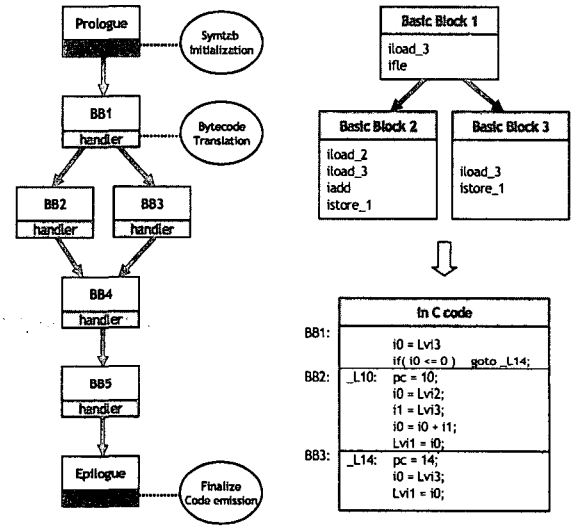

Fig.2. Overview of bytecode-to-C translation process.

2.5 Runtime Libraries

Java platform includes several API's which are able to invoke native methods. Unfortunately the implementation of native methods can be changed according to a target system. So whenever the target system changes, native methods will be directly coded again. Our Java-to-C compiler fully supports Connected Limited Device Configuration 1.0 (CLDC 1.0) API that includes native methods. In order to simplify the generated C codes, some runtime routines were added to runtime library.

3. Performance Evaluation

3.1 Methodology

The performance of our Java-to-C compiler has been tested using Java SciMark 2.0 benchmarks[5]. We used the following systems to compare the performance with ours : Sun's java interpreter (JDK1.2.2), Tya1.7 JIT compiler[6] and Shujit JIT compiler[7]. We measured the performance on an

workstation with two Zeon 2.0 GHz processors and 512 MB of memory. The benchmarks were tested(executed) on Linux Redhat 9.0.

3.2 Performance

The relative speedup to the JVM-only execution (JVM 1.2.2) on three systems is shown in Figure 3. Our Java-to-C compiler shows better performance than other systems for all applications except MonteCarlo. The executable code that was generated from Java FFT code by our Java-to-C compiler is about eleven times as fast as the code on JVM 1.2.2. It means that the our compiler performs the following operations very quickly : complex arithmetic, shuffling, non-constant memory references and trigonometric functions. Others also could be interpreted with the same scheme.

The overhead for synchronization in single-threaded application makes that the our system has poor performance for Monte Carlo Integration than the others. In our compiler, even if the application is single-threaded, monitor locking is enabled. Toba[3] can reduce the synchronization overhead, since the actual monitor locking is delayed until more than one thread will be created.

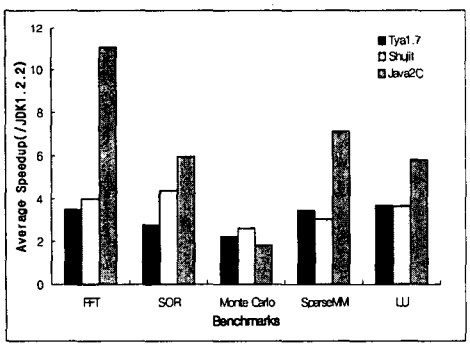

Fig.3. Java SciMark 2.0 speedup.

3.3 Aggressive Exception Handling

Our Java-to-C compiler has a flexible code generation framework. So we can improve the performance without any additional effort. Figure 4 shows that the elimination of the system defined exception handling routines will improve the performance of the generated code by 1.2 to 3 times.

4. Conclusion

We represent that the Java-to-C compiler can attain better performance than other approaches, especially the JIT compilers. The features for mobile computing are implemented in our Java-to-C compiler makes it will apparently be distinguished with the existing Java-to-C compilers especially Toba[3].

The generated C code include many pointer and complex expressions, which prevent AOT compilers from applying advanced compiler optimization techniques like constant propagation, subexpression elimination, inlining, and so on. In the ongoing research, we have developed an IR framework between bytecodes and C codes for helping AOT compiler generate better quality of codes.

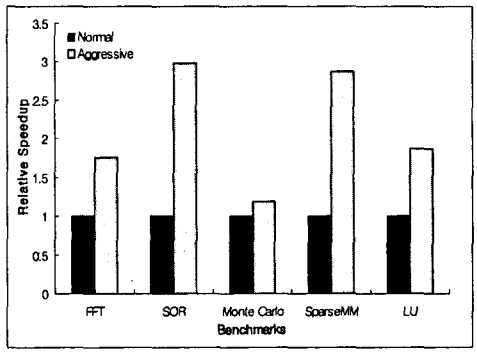

Fig.4. Speedup by an aggressive excpetion handling

Reference

1. Wireless Internet Platform for Interoperability(WIPI). http://wipi.or.kr
2. GNU Virtual machine(GVM). http://www.gvmclub.com
3. Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (WAT) compiler. In Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS97), June 1997.
4. Ankush Varma and Shuvra S. Bhattacharyya. *Java-through-C compilation*: An enabling technology for java in embedded systems. In Design Automation and Test in Europe (DATE03), Paris, France, February 2004.
5. SciMark 2.0. http://math.nist.gov/scimark2/
6. Tya JIT Compiler. http://sax.sax.de/~adlibit/
7. shuJIT Compiler. http://www.shudo.net/jit/%#docs