Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results

Cheng-Hsueh A. Hsieh   John C. Gyllenhaal   Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
ada, gyllen, hwu@crhc.uiuc.edu

Abstract

The Java bytecode language is emerging as a software distribution standard. With major vendors committed to porting the Java run-time environment to their platforms, programs in Java bytecode are expected to run without modification on multiple platforms. These first generation run-time environments rely on an interpreter to bridge the gap between the bytecode instructions and the native hardware. This interpreter approach is sufficient for specialized applications such as Internet browsers where application performance is often limited by network delays rather than processor speed. It is, however, not sufficient for executing general applications distributed in Java bytecode. This paper presents our initial prototyping experience with Caffeine, an optimizing translator from Java bytecode to native machine code. We discuss the major technical issues involved in stack to register mapping, run-time memory structure mapping, and exception handlers. Encouraging initial results based on our X86 port are presented.

1. Introduction

The software community has long desired a universal software distribution language. If such a language is widely supported across systems, software vendors can compile and validate their software products once in this distribution language, rather than repeating the process for multiple platforms. Software complexity is rapidly increasing and validation has become the deciding factor in software cost and time to market. Therefore, substantial economic motivation exists behind efforts to create such a software distribution language. The progress, however, has been very slow due to legal and technical difficulties.

On the legal side, many software vendors have been skeptical about the ability of the proposed software distribution languages to protect their intellectual property. In practice, such concern may have to be addressed empirically after a standard emerges. Although the protection of intellectual property in software distribution languages is an intriguing issue, it is not the topic addressed by this paper. For the purpose of our work, we expect Java to be accepted by a sufficient number of software vendors in the near future to make our work relevant.

On the technical side, the performance of programs distributed in a universal software distribution language has been a major concern. The problem lies in the mismatch between the virtual machine assumed by the software distribution language and the native machine architecture. The task of bridging the gap is made more difficult by the lack of source code information in the distributed code in order to protect intellectual property. As a result, software interpreters have been the main execution vehicles in the proposed standards. The disadvantage of software interpreters is poor performance. This disadvantage has been partially compensated for by the fast advance of microprocessor speed. For applications such as Internet browser applets where overall performance is often more limited by network delays than processor speed, sacrificing processor performance in favor of reducing software cost has become acceptable. This is, however, not true for general applications.

This paper presents our initial prototyping experience with *Caffeine,* an optimizing Java bytecode to native machine code translator. Although our techniques are presented in the context of handling Java, they are applicable to other software distribution languages such as Visual Basic P-code. We are by no means arguing that Java is the ultimate software distribution language. Rather, we intend to develop a strong portfolio of techniques from our Java implementation efforts that will contribute to the creation and acceptance of whatever language becomes the final standard. The objective of this work is to run the translated code at nearly the full performance of native code directly generated from a source representation such as the C programming language.

Due to space limitations, we will limit our discussion to three critical issues involved in the translation process. The first issue is the mapping of the stack computation model of the bytecode Virtual Machine to the register computation model of modern processors. A performance enhancing algorithm that takes advantage of the register computation model is presented. This algorithm requires analysis to identify the precise stack pointer contents at every point of the program. In addition, most compilation infrastructures require that each virtual register contains just one type of data, and that virtual registers do not overlap. We present a live-range-based register-renaming algorithm that can resolve such inconsistencies in non-pathological cases. The second issue is mapping the bytecode memory organization to the native architecture. A more efficient memory organization than the one used by the Java interpreter is introduced. The third issue is how to translate the exception handling semantics of Java. We describe the preliminary method used by *Caffeine* and some of the issues involved.

A prototype of *Caffeine* has been developed based on the IMPACT compilation infrastructure [1]. The prototype is sufficiently stable to handle Java bytecode programs of substantial size. This paper presents some initial experiments comparing the real machine execution time of Java bytecode programs using the SUN Java interpreter 1.0.2, the Symantec Java Just-in-time (JIT) compiler 1.0, and the IMPACT Java to X86 native code translator 1.0 running under Windows 95. Also included in the comparison is the execution time of equivalent C programs directly compiled by the Microsoft Visual C/C++ compiler 4.0 into X86 native code. Preliminary results show that the optimizing translator is currently capable of achieving, on average, 68% of the speed of the directly compiled native code.

The remaining sections are organized as follows. Section 2 introduces different approaches to execute Java bytecode programs and an overview of our translation steps. Section 3 presents the stack computation model used by Java followed by a proposed stack to register mapping. An overview of the stack analyses required to perform and validate this mapping is introduced in Section 4. Section 5 discusses the run-time memory model adopted by the SUN Java interpreter and presents a more efficient organization. Complications due to exception handling are discussed in Section 6. Preliminary performance results are presented in Section 7. Section 8 provides some concluding remarks and directions of future work.

2. Background

We will not cover the Java bytecode Virtual Machine model in this paper due to space limitations. Interested readers are referred to the Java web site [2] and a large collection of Java literature [3-11]. We will instead introduce three competing and sometimes complementary approaches to execute Java bytecode programs: interpreters, just-in-time compilers, and optimizing native code translators. A preliminary performance comparison between these approaches and native code execution are presented in Section 7.

Interpreters are the most widely understood approach to execute Java bytecode programs. A software interpreter emulates the Java bytecode Virtual Machine by fetching, decoding, and executing bytecode instructions. In the process, it faithfully maintains the contents of the computation stack, local memory state, and structure memory. The Java interpreter from SUN Microsystems is available to the public [2].

Just-in-time compilers do on-the-fly code generation and cache the native code sequences to speed up the processing of the original bytecode sequences in the future. The current generations of just-in-time compilers do not save the native code sequences in external files for future invocations of the same program. Rather, they keep the native code sequence to speed up the handling of the corresponding bytecode sequence during the same invocation of the program. Thus, they take advantage of iterative execution patterns such as loops and recursion. At the time of this work, Borland [12] and Symantec [13] had both announced just-in-time compiler products, and the Symantec JIT compiler is used in this paper. Due to the code generation overhead that occurs during program execution, just-in-time compilers are still intrinsically slower than executing native code programs.

Optimizing native code translators use compiler analysis to translate bytecode programs into native code programs off-line. This is the least understood approach among the three alternatives. Without extensive analysis and transformation capabilities, the native code generated may not be much better than that cached in the just-in-time compilers. Therefore, optimizing native code translators must perform extensive analysis and optimization in order to offer value beyond just-in-time compilers. Such analysis and transformations tend to make the translation process more expensive in time and space. In general, only those applications that will be repeatedly invoked or those applications whose execution time is much longer than the translation time should be translated. Thus, optimizing native code translators will not eliminate the need for interpreters and just-in-time compilers.

Figure 1 shows an overview of the steps in our prototype optimizing native code translator. The Java class files [4] required to execute the program are identified and decoded into sequences of bytecode operations, which are later used for construction of an internal representation (IR), called the Java IR, which is organized into functions and basic blocks. The construction of the Java IR is straightforward due to the absence of indirect jumps, indirect calls, self-modified code, embedded data, and branch target alignment “filler” code in bytecode. Due to the nature of the Java Virtual Machine specification [4] and the class file format, data recognition is also straightforward. Thus, the information recovered from Java bytecode ensures complete control flow graph construction.

The IMPACT low-level intermediate code (Lcode) serves as a machine-independent IR for our prototype translator. Translation from the Java IR to an efficient Lcode IR requires extensive analyses, as discussed in Section 4. The stack computation model is mapped to a more efficient register computation model. Bytecode operations which do not have corresponding Lcode operations are translated into sequences of

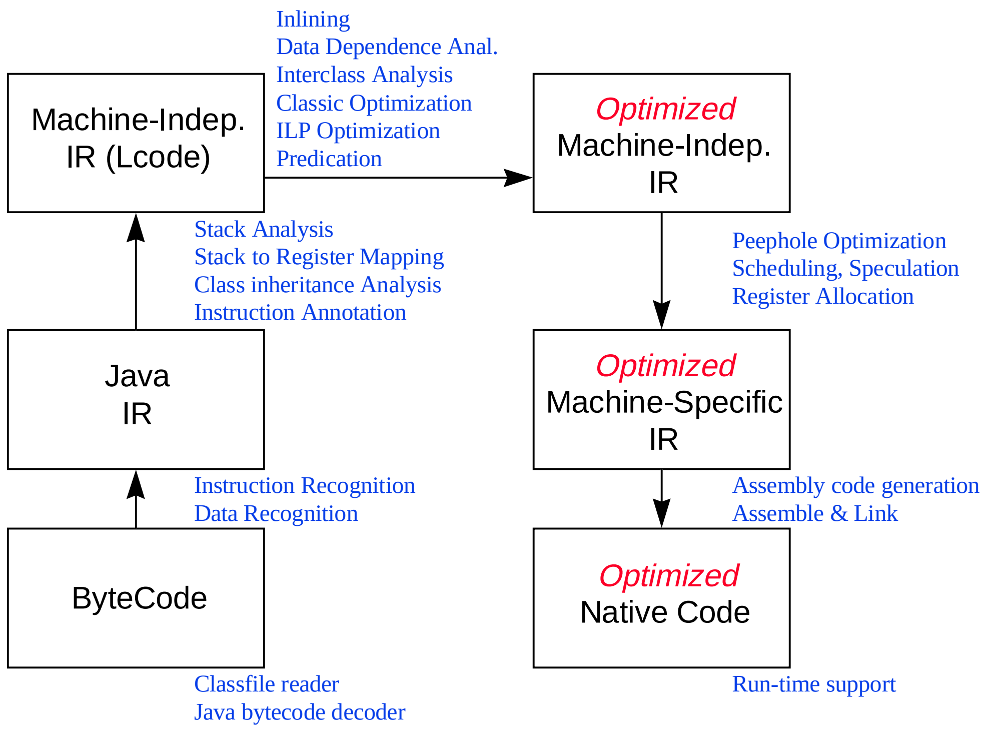

Figure 1. Java bytecode to native code translation steps.

Stack operations	Translated code
push A	push A
push B	push B
add	r2 ← pop (B)
	r1 ← pop (A)
	r3 ← r1 + r2
	push r3

Figure 2. Translated intermediate code example without stack to register mapping.

Lcode operations or into function calls to the emulation library. After the Lcode IR is constructed, it is optimized by the X86 compilation path in the IMPACT compiler to generate assembly code and then an executable which runs under Windows 95. The Lcode IR construction phase is generic and will be retargeted to other code generation paths supported in IMPACT compiler in the future.

3. Stack to Virtual Register Mapping

3.1 Stack Computation Model

Java bytecode Virtual Machine uses a stack computation model to avoid making assumptions about the architectural register file size available to the interpreter [4]. Source operands are fetched from the top of *operand stack* and the result is pushed back on. The instruction size in this model is small since the operands are implicitly defined and require no operand fields in the instruction encoding, which facilitates efficient object code distribution over the Internet. Beside the operand stack, the Java Virtual Machine also provides a memory array, called the *local variable array*, for storage of local variables.

No stack analysis is required if the translated native code maintains a run-time operand stack in memory and manipulates it in the same way that the interpreter does. This straightforward approach is able to handle any situation that the interpreter can handle. The run-time cost of this straightforward approach, however, can be expensive due to the unnecessary memory traffic caused by inefficient register utilization. In Figure 2, the stack operations and the corresponding unoptimized translated intermediate code are presented side-by-side for an add operation to illustrate this approach. A load/store architecture is assumed in this example. Note that the original add operation pops two operands off the stack, adds them, and pushes the result back on.

Optimizations can be performed on the translated code to eliminate some of the loads (pops) and stores (pushes). However, many will still exist due to the use of stack operations across basic blocks. Global removal of unnecessary loads and stores requires an analysis equivalent to that discussed in Section 4 and is not the focus of this paper.

3.2 Register Mapping: Global Stack Location Register Mapping and Renaming

The performance of the translated code can be improved by mapping the run-time stack to the virtual register file. The approach used by *Caffeine* is to assign each stack location a unique virtual register number. Register allocation is later used

Stack operation	Translated code	After copy prop. & Dead code removal
push A	r1 🡐 A	-
push B	r2 🡐 B	-
add	r1 🡐 r1 add r2	r1 🡐 A add B

Figure 3. Translated intermediate code example with stack to register mapping.

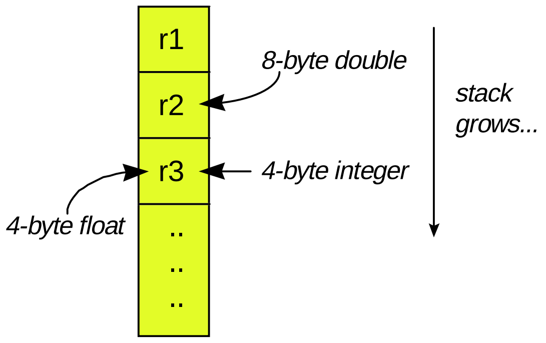

Figure 4. Example of type or size mismatch and register aliasing problems. A 32-bit architecture is assumed.

to allocate the virtual registers to physical registers during the code generation phase. After this register mapping, a *push* to the operand stack is translated to a *move* to the register assigned to the stack location pointed to by the current stack pointer, and a *pop* is translated to a *move* from the register assigned to the stack location. This algorithm can only be applied when a constant stack offset can be determined for every push and pop at translation time. Algorithms to determine when this transformation can be applied are discussed in Section 4.1. The local variable array can be mapped to virtual registers using the array indices. Figure 3 shows the translated code using this approach for the same add operation as Figure 2. The moves of operands to virtual registers r1 and r2 before add will be forward copy propagated if possible. They are then removed as dead code if they are not live out.

There are two issues associated with this approach that need to be resolved. First, variables with different types or different sizes may be pushed to the same stack location and thus assigned to the same virtual register, causing some virtual registers to hold multiple types of operands or to alias with adjacent registers. In Figure 4, a push of 4-byte float onto stack location 3 is translated to "*r3* 🡐 *float_value*" in this register mapping scheme. At another point in the program, a 4-byte integer could be pushed to the same stack location and be translated to "*r3* 🡐 *integer_value*". As a result, register *r3* holds two different types, which is not allowed in many compiler infrastructures. Another conflict arises if an 8-byte double is pushed to stack location 2 and 3. The translated statement "*r2* 🡐 *double_value*" causes register *r2* to alias with

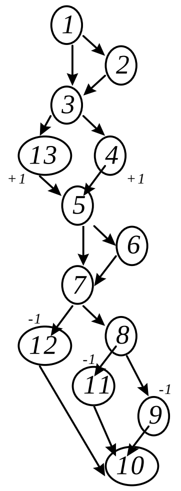

Figure 5. Stack balance analysis example.

neighboring register *r3*. Because Java has no union and all type conversions are made explicit, accesses to different types should never alias. Bytecode generated from a valid Java compiler should always have the *type state* property [6] that guarantees neither type conflict nor aliasing problems should occur. An algorithm presented in Section 4.2 is used to validate this assumption and to disambiguate virtual registers which hold different types in this mapping scheme.

Second, parallelism may be lost for wide-issue machines because different variables use the same stack location in the original Java bytecode and get assigned to the same virtual register in this mapping scheme. This reuse of the virtual registers introduces artificial output and anti-dependencies. The same algorithm used to disambiguate virtual registers which hold different types can be applied to perform global virtual register renaming to remove the artificial dependencies.

4. Stack analysis

4.1 Stack balance analysis

For our register mapping scheme to function correctly, the position of the stack pointer must be a known constant for each operation at translation time. Although bytecode generated from valid Java compilers should satisfy this property [6], we can not assume all loaded bytecode came from valid sources.

A basic block may push more items on the stack than it consumes, and vice versa. The *residue* of each basic block, which is defined as the total number of pushes minus the total number of pops in the block, is computed first. The control flow graph is then traversed depth-first and each node is marked "visited" along the path from the first block. The stack pointer position upon entering a block is equal to the accumulated residue. If a marked basic block is revisited, the accumulated residue is checked against the stack pointer position in the revisited block. If they disagree, the stack to register mapping cannot be applied to this control flow graph. In such cases, *Caffeine* reverts to the stack model. The accumulated residue is also checked against zero whenever a

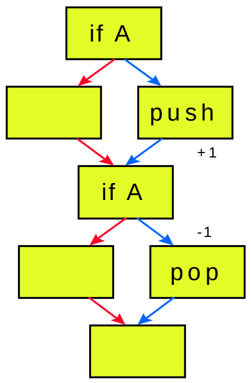

Figure 6. Example of where stack based approach must be used.

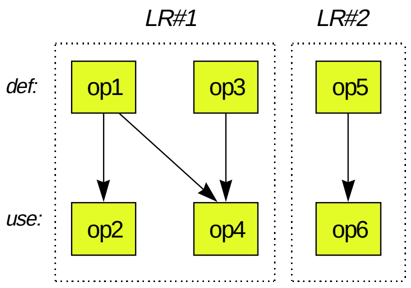

Figure 7. Def-use chains grouping.

leaf block (a block with no successor) is reached to ensure that the stack is balanced in each control flow graph. This algorithm runs in linear time in the number of basic blocks and control flow arcs. Figure 5 shows a control flow graph whose blocks are numbered in the order that they are visited by this algorithm. In this example, we assume blocks 4 and 13 have a residue of one, and blocks 9, 11, and 12 have a residue of minus one. All paths from block 1 to block 10 are stack balanced. The position of the stack pointer for each block is also a known constant. Specifically, the stack pointer at the entrance to blocks 5, 6, 7, 8, 9, 11, and 12 points to location one. For the rest of the blocks, the stack pointer initially points to location zero.

An example of when register mapping cannot be currently applied is shown in Figure 6. Depending on the path traversed, the stack offset for the pop is either zero or one. For this case, the stack based method needs to be used. However, since we are using a valid Java compiler, the register-based approach can always be used.

4.2 Live range disambiguation and register renaming

The mapping of stack locations and local variables to registers could have type and size conflicts as discussed in Section 3.2. Variables of different types which reside in the same virtual register are separated into separate registers, when

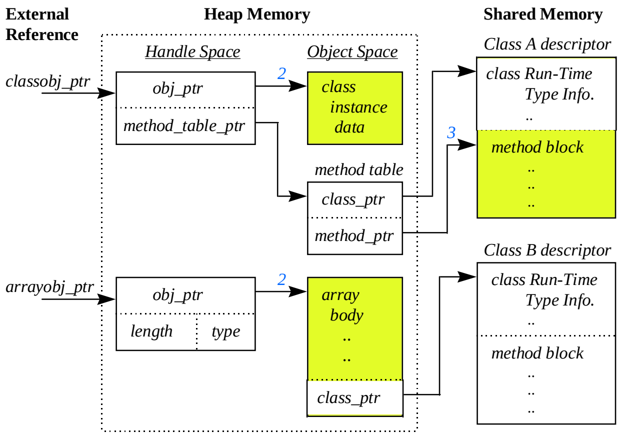

Figure 8. Run-time memory organization used by Java interpreter.

possible, using the technique presented below. The live ranges of each register can be characterized by their def-use chains. Since the access of a double also takes the next contiguous memory word, the normal reaching-definition analysis [16] is slightly modified to take this effect into account. To be specific, the definition of *r2* by an 8-byte double in Figure 4 also reaches the contiguous register *r3*.

For each register *rx*, the identified def-use chains are grouped into non-overlapping live ranges. In Figure 7, operations op1, op3, and op5 define the register *rx* and operations op2, op4, and op6 use the same register *rx*. The def-use chains for register *rx* are 1→ 2, 1→ 4, 3→ 4, and 5→ 6 as shown. Each connected graph forms a non-overlapping live range of register *rx*. As a result, op1, op2, op3, and op4 form a live range (LR#1) while op5 and op6 form another (LR#2). Register *rx* in each live range is renamed to a different register id. If the type of register *rx* is not consistent inside a live range after this renaming, the stack to register mapping cannot be applied and the translation falls back to the stack computation model.

5. Run-time memory organization

Java programs are name-binding rather than address-binding and thus allow flexibility in the run-time memory organization implemented by the interpreter. Dynamically allocated objects in the heap can be roughly categorized into *class* objects and *array* objects. Figure 8 illustrates the heap memory organization used by the SUN Java interpreter. In this organization, neither a class object nor an array object points directly to its associated data. Rather, there is an 8-byte *handle* in between. Accesses to both *class instance data* and *array body* require two levels of indirection. Accesses to the *method block* for method invocation need three levels of indirection. Since these access events take place frequently during program execution, such high levels of indirection can cause significant performance degradation.

The enhanced memory model proposed in this paper (in Figure 9) reduces the amount of indirection by combining the

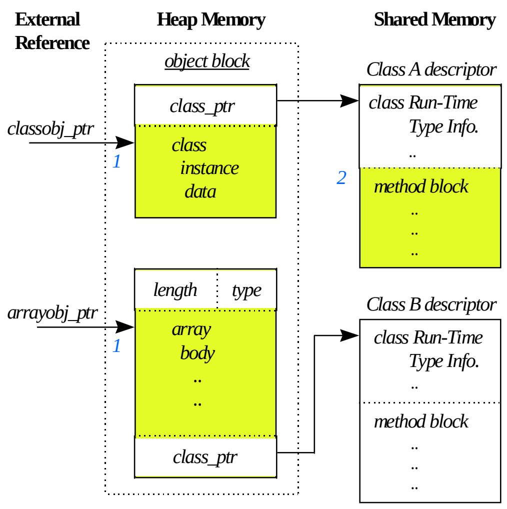

Figure 9. Run-time memory organization used by Java bytecode translator.

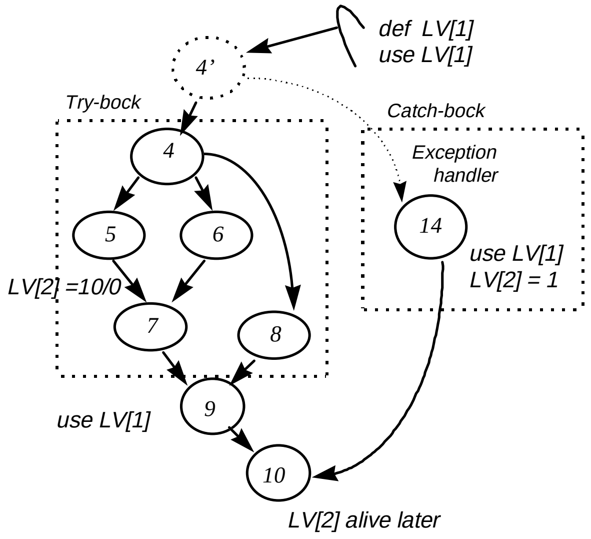

Figure 10. Example of exception handing.

class instance data block and the *method table* into one *object block*. The reference to *object block* now requires only one level of indirection. Since the class run-time type information in our implementation is of constant size, the method block can be accessed by a constant offset from a pointer to the class descriptor. The *method_ptr* in Figure 8 is thus eliminated, which reduces a method block reference to two indirection levels. The enhanced model also consumes less memory. Changes made to the run-time library, which is source licensed from SUN, to support this enhanced memory model are minimal due to the library's heavy use of preprocessor macros for handle-to-object dereferencing.

6. Exception Handler Considerations

Exception handlers are sections of code that are reached when a run-time exception occurs. The *try*-block in Java is designed to enclose statements which may cause run-time exceptions. Exceptions which occur within a try-block are captured by an associated *catch*-block of the same exception type. A Java *method* can have many exception handlers cascaded together to guard ordinary code, or to guard other handlers. In Figure 10, block 14 is an exception handler that guards its try-block consisting of blocks 4 to 8. There are four issues that must be addressed during translation.

First, after exception handling, control may be transferred back to the original program (e.g. in Figure 10, block 14 → 10). As a result, exception handlers need to be connected to the control flow graph as shown in Figure 10.

Second, an exception handler might use local variables defined before its associated try-block. In Figure 10, the definition of local variable entry *LV[1]* reaches the use in exception handler block 14. A pseudo arc, shown as a dotted line, and a null block preceding the try-block are created to allow live variable information to be passed to the handler. Otherwise incomplete flow analysis may lead to incorrect optimizations.

Third, during optimization and scheduling, an instruction inside the try-block cannot be moved outside its try-block without enlarging the try-block in general. However, if the try-block has to be enlarged, to avoid changing the program behavior, the added instructions should not cause exceptions that can be captured by the try-block's handler.

Fourth, for maximum portability, exception handling support in the Java interpreter does not rely on the underlying architecture or operating system. Thus, the interpreter explicitly checks for null references, array index bounds, divide by zero, etc. It is expensive and often unnecessary for the translated code to do all of these explicit checks. *Caffeine* currently explicitly checks array index bounds in the translated code. This checking costs about 10% of the performance across our benchmark programs. Optimization opportunities exist to conduct analysis to eliminate unnecessary explicit checks. Previous work has shown that program analysis can be done to determine if it is possible for a load or store to ever have an address of zero or to ever access outside of its intended array, etc., for the purpose of speculative code motion [20].

The benchmarks presented in Section 7 do not cause exceptions and thus do not exercise the exception handler capabilities of Java. Although *Caffeine* does not currently support many of these capabilities, we believe that the underlying hardware architectures can be used to support the remaining exception-handling capabilities without affecting performance of these benchmarks.

7. Benchmarks and Preliminary Results

A suite of six integer programs was selected to evaluate our prototype translator. There were currently no standard Java benchmarks generally available at the time of this work. For each program, we hand translated the C source code into equivalent Java source code. By equivalence we mean that the

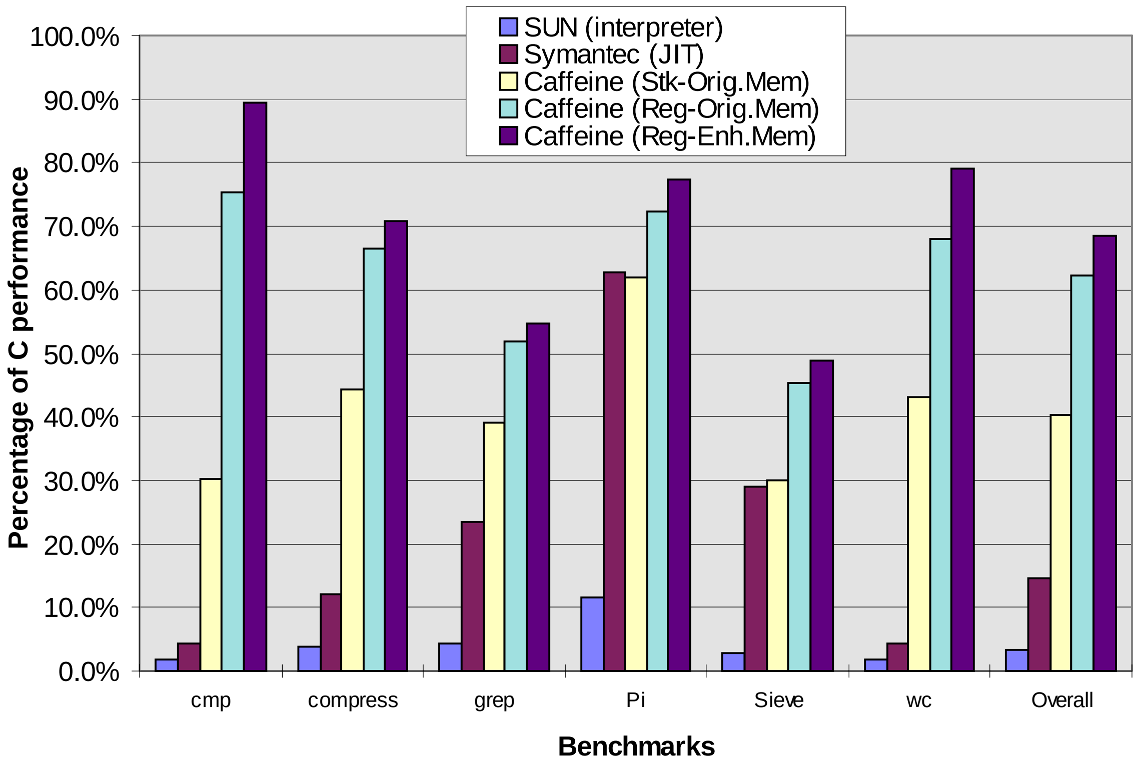

Figure 11. Experiment results on different approaches. All numbers are relative speed to the equivalent C code compiled by Microsoft Visual C/C++ compiler with optimization level two.

algorithm, data structures, and operand types used in the Java code and the C code are the same. Due to the fundamental differences between C and Java with regard to the object-oriented concept, array accessing, array index bounds checking and library routines, an exact correspondence is not always feasible. When this occurred, we modified the C program so that it could be translated with close correspondence. The Java sources thus generated are then compiled into Java bytecode by the SUN Java compiler.

Figure 11 shows preliminary results that compare the real machine execution time of Java bytecode programs using the SUN Java interpreter, the Symantec Java Just-in-time compiler (JIT), and different configurations of the IMPACT Java to X86 native code translator *Caffeine*. All of the programs are executed on an Intel Pentium processor running Windows 95. Performance is shown in Figure 11 as a percentage of the benchmark performance for the equivalent C code compiled by the Microsoft Visual C/C++ compiler with optimization level two. The first *Caffeine* model (Stk.-Orig.Mem) uses the stack computation model and the interpreter's memory model. The performance is, on average, 2.8 times higher than the JIT compiler. This is because of the optimizations that remove unnecessary pushes and pops, and because no initial code-generation is required. The second *Caffeine* model (Reg.-Orig.Mem) uses the register computation model instead of the stack model. This results in 55% performance improvement over the stack model. The final *Caffeine* model (Reg.-Enh.Mem) also uses the proposed memory organization instead of the interpreter's memory organization. This results is a 7% performance improvement. This final model of our prototype Java native code translator *Caffeine* is capable of generating code that runs on average at 68% of the speed of the equivalent C code, 4.7 times faster than the Symantec Java JIT compiler, and more than 20 times faster than the Java interpreter. For these preliminary results, the *Caffeine* translated code is optimized using classic C code optimization techniques without profiling and inlining.

8. Conclusion and Future Work

In this paper, we presented our initial prototyping experience with *Caffeine*, a Java-bytecode-to-native-machine-code translator, to demonstrate the feasibility of efficient universal software distribution languages. The preliminary results show that it is capable of achieving 68% of the speed of the native code directly compiled from the equivalent C code. Besides the fact that it removes the interpretation overhead, much of the performance gain over the SUN Java interpreter comes from the stack to register mapping, which fully utilizes the register computation model of modern processors. The requirements and algorithms for the stack to register mapping were presented and discussed. Although these requirements will hold for all Java bytecode generated by a valid Java compiler, the stack computation model is kept as a fall-back when these requirements are not met. The penalty for using the stack model is about a 35% performance degradation.

We also presented and compared two different run-time memory organizations. Preliminary results showed that a 7% performance gain can be achieved by moving the data associated with dynamically allocated objects closer to their external references.

Several aspects of translating Java bytecode to native code that were not exercised by these benchmarks are now being investigated. These aspects include garbage collection, Java's extensive exception handling capabilities, threading support, and the use of the Java graphic library.

In addition, substantial ongoing efforts are focusing on removing indirection overhead for method invocation. By doing interclass reaching-definition analysis, we should be able to trace the class type of a current object from its definition and convert, if possible, the indirect method invocation to an absolute method invocation. Inlining is also made possible by this conversion. Another direction for research is to perform better memory disambiguation by taking advantage of well-protected class boundaries to eliminate dereferencing overhead. We also observe that the array index bounds checking as required by Java semantics is a major source of performance degradation. We feel that with aggressive analysis, most of these checks can be removed. We would also like to target other platforms and make use of more advanced instruction-level parallelism enhancing techniques such as predication and speculation.

Acknowledgments

The author would like to thank Daniel M. Lavery for proofreading the various versions of this paper, all the members of the IMPACT research group, and the anonymous reviewers whose comments and suggestions helped to improve the quality of this paper.

This research has been supported by the National Science Foundation (NSF) under grant MIP-9308013, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

Reference

[1] P.P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, *IMPACT: An architectural framework for multiple-instruction-issue processors*, Proc. 18th Ann. Int'l Symp. Computer Architecture, (Toronto, Canada), pp. 266-275, Jun 1991.

[2] *Java™ – Programming for the Internet,* Sun Microsystems, Inc., 1996, *http://java.sun.com/*

[3] James Gosling and Henry McGilton, *The Java Language Environment, A White Paper*, Sun Microsystems Computer Corporation, October, 1995.

[4] *The Java Virtual Machine Specification, Release 1.0 Beta DRAFT*, Sun Microsystems Computer Corporation, August 21, 1995.

[5] *The Java Language Specification, Version 1.0 Beta DRAFT*, Sun Microsystems Computer Corporation, October 30, 1995.

[6] James Gosling, *Java intermediate Bytecodes*, ACM SIGPLAN Workshop on Intermediate Representations, 1995.

[7] Arthur van Hoff, Sami Shaio, and Orca Starbuck, *Hooked on Java*, Addison-Wesley, December 1995.

[8] David Flanagan, *Java in a Nutshell*, O'Reilly & Associates, Inc, February 1996.

[9] Gary Cornell and Cay S. Horstmann, *Core Java*, The Sunsoft Press Java Series, March 1996.

[10] Michael C. Daconta, *Java for C/C++ Programmers*, Wiley Computer Publishing, March 1996.

[11] Ken Arnold and James Gosling, *The Java Programming Language*, Addison Wesley, May 1996.

[12] *Borland C++ Development Suite*, Borland International, Inc., 1996 , http://www.borland.com/

[13] *Café – Visual Java Development and Debugging Tools*, Symantec Corporation, 1996, http://www.symantec.com/

[14] Tim Wilkinson, *KAFFE – A JIT virtual machine to run Java code*, 1996, *http://web.soi.city.ac.uk/homes/tim/kaffe/kaffe.html*

[15] *Guava – High-performance Environment for Running Java Programs*, Softway Pty. Ltd., 1996, *http://www.softway.com.au/softway/products/guava/*

[16] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman, *Compiler – Principles, Techniques, and Tools*, Addison Wesley, March, 1988.

[17] Jeffery Richter, *Advanced Windows - Chap.9 Thread Synchronization and Chap.14 Structured Exception Handling*, Microsoft Press, 1995.

[18] Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books Worldwide, 1995.

[19] Walter Oney, *Extend Your Application with Dynamically Loaded VxDs Under Windows 95*, MSJ, May 1995.

[20] Roger Alexander Bringmann, *Enhancing Instruction Level Parallelism Through Compiler-Controlled Speculation*, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995.