Using *NET* to Capture Performance in Java-Based Software

Cheng-Hsueh A. Hsieh Marie T. Conte Teresa L. Johnson John C. Gyllenhaal
Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
University of Illinois
1308 W. Main St.
Urbana-Champaign, IL 61801
ada, mconte, tjohnson, gyllen, hwu@crhc.uiuc.edu

Cheng-Hsueh A. Hsieh	(408) 929-1150
Marie T. Conte	(217) 244-1277 Fax: (217) 333-5579 (coordinating author)
Teresa L. Johnson	(217) 244-1081
John C. Gyllenhaal	(217) 244-3226
Wen-mei W. Hwu	(217) 244-8270

Keywords: Universal Software Distribution Environment
Universal Software Distribution Language
Optimizing Java Bytecode Translator
Stack to Register Mapping
Java Memory Organization
Java Verification Costs
Java Garbage Collection
Java Cache Performance
Java Array Index Bounds Checking

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. These first generation run-time environments rely on an interpreter, possibly extended with capabilities to cache native code for reduced interpreter overhead, to bridge the gap between the bytecode instructions and the native hardware. The 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 article presents our initial prototyping experience with our Native Executable Translation (NET) compiler, an optimizing Java-bytecode-to-native-machine-code translator. We discuss the major issues involved in improving run-time performance as well as some less obvious costs. Encouraging initial results based on our X86 port are presented.

1. Introduction

Java, the new object-oriented programming language from Sun Microsystems, appears to be setting the standard for universal software development[1]. Java code compiles into a binary format called bytecode which can be used for software distribution, and which does not need recompilation in order to run on any platform. Java is also secure, guarding against code corruption before execution[2]. This new language uses run-time resolution to locate objects and their corresponding classes, meaning software updates can be integrated as quickly as they are made available[1]. However, there are several trade-offs and costs involved in migrating to a universal software distribution environment.

Currently there are four approaches to running Java: an interpreter, a Just-In-Time (JIT) compiler, a Native Executable Translation (NET) compiler, and a hardware implementation. In this article we present our initial prototyping experience with our NET compiler, an optimizing

Java-bytecode-to-native-machine-code translator. 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/C++ programming languages. However, the work with our NET compiler is not limited to Java. Our goal is to develop a strong portfolio of techniques from our Java implementation efforts that will contribute to the development and acceptance of any universal software distribution language.

Due to space limitations we will focus our discussion on the critical issues involved in the design of our NET compiler. However, we also explain some of the less intuitive costs involved in running Java. The critical issues include: minimizing verification overhead, mapping the stack computation model of the bytecode virtual machine to the register computation model of modern processors and developing a more efficient memory organization. We also present some preliminary results for several large application programs and standard benchmarks, running on a 166MHz Pentium system, and we compare the NET compiler translated code performance with Sun's Java Virtual Machine version 1.0.2 and Microsoft's Java Just-In-Time compiler version 1.0[3]. Also included in the comparison is the execution time of equivalent C/C++ programs directly compiled by the Microsoft Visual C/C++ compiler version 4.2. Preliminary results show that our optimizing NET compiler is currently capable of achieving better performance than the other bytecode execution methods, in some cases achieving speeds comparable to directly compiled native code[4].

2. Overview and Motivation

Our interest in Java is motivated by four key aspects of Java bytecode-based software distribution. First, Java adds security to the distribution of software, providing added benefits over more common languages like C/C++. Second, the Java Virtual Machine (VM) defines an interface for executing Java bytecode programs on a wide variety of vendor platforms, allowing for the development of a truly portable software base. Third, Java contains many features that we believe

are fundamental to the success of any universal language. Last, due to the nature of interpreted languages, Java executables run slower than their compiled counterparts in other languages. Our objective is to develop a technology for executing Java bytecode programs at high performance while fully supporting all of the desirable Java features. In the remainder of this section we describe the three software models for running Java bytecode: the interpreter, the JIT compiler, and our NET compiler.

Interpreters are the most widely available approach to executing Java bytecode programs. A software interpreter emulates the Java VM 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 publicly available[5].

Just-In-Time (JIT) compilers perform on-the-fly code generation of frequently executed Java methods, while emulating the VM, and cache the native code sequences to speed up the processing of the original bytecode sequences in the near 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 retain cached native code sequences to speed up the corresponding bytecode sequences during the same invocation of the program. At the time of this work, Borland[6], Symantec[7], and Microsoft[3] have all released Just-In-Time compiler products, and the Microsoft JIT compiler is used in this article. Due to the code generation overhead that occurs during program execution, and limited time available to perform code optimizations, Just-In-Time compilers are still intrinsically slower than direct native code execution. In addition, due to the need to explicitly generate and cache native code, this approach requires more effort than interpreters when porting to new platforms.

Native Executable Translation (NET) Compilers use state-of-the-art compiler analysis to translate bytecode programs into native code programs off-line. The fundamental difference

between a JIT and a NET compiler is that the code compiled by a NET is intended to be saved for future invocations, under protection of an updater to support software updates. However, without extensive analysis and optimization capabilities, the native code generated may not perform much better than that cached by JIT compilers. Such analysis and transformations tend to make the translation process more expensive in terms of time and space. In general, only those applications that are repeatedly invoked or those applications for which the execution time is much longer than the translation time should be translated. Thus, optimizing NET compilers are unlikely to fully eliminate the need for interpreters and JIT compilers. NET compilers are also the least understood approach among the three alternatives, and require significantly more effort to port to new platforms than interpreters.

Figure 1 shows an overview of the steps in our prototype optimizing NET compiler. The Java class files required to execute the program are identified, parsed, and translated into an internal representation (IR), called the Java IR, consisting of methods and basic blocks of bytecode instructions. Then, several optimization techniques are applied before generating the final native code output. Our NET compiler is based on the IMPACT compilation infrastructure[8]. The prototype is sufficiently stable to handle Java bytecode programs of substantial size.

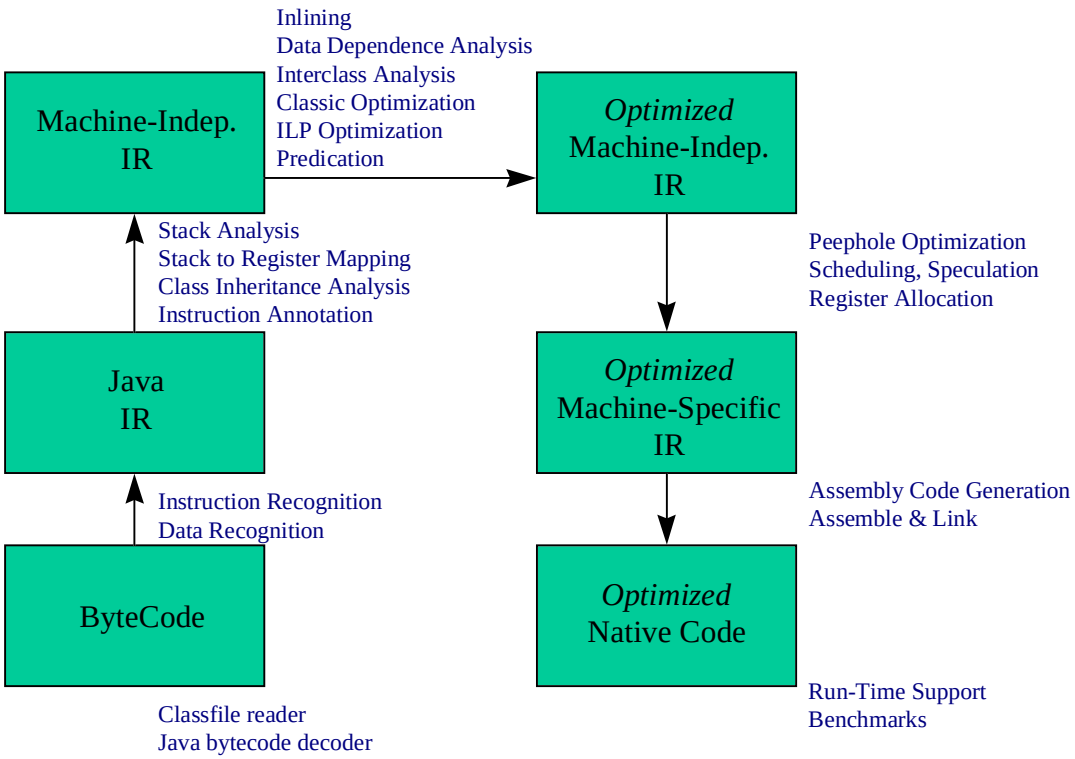

Figure 1: Overview of the IMPACT NET Compiler Translation Path

3. Benefits of Translating to Native Code

Among the reasons for Java's success are features such as the ease of software updates, an architecture-independent implementation, and verification support. Unfortunately, these features also contribute significantly to the performance penalty experienced when running Java, particularly with an interpreter. Our NET compiler translates Java bytecode directly into machine code, reducing the cost of these features and enabling aggressive optimizations (see Figure 1). Furthermore, we believe the translated code can eventually achieve performance very close to C/C++ programs compiled directly for the underlying architectures, while maintaining the security and other added benefits associated with Java. Figure 2 shows an overview of the IMPACT NET compiler framework, and how it fits into the Java system architecture.

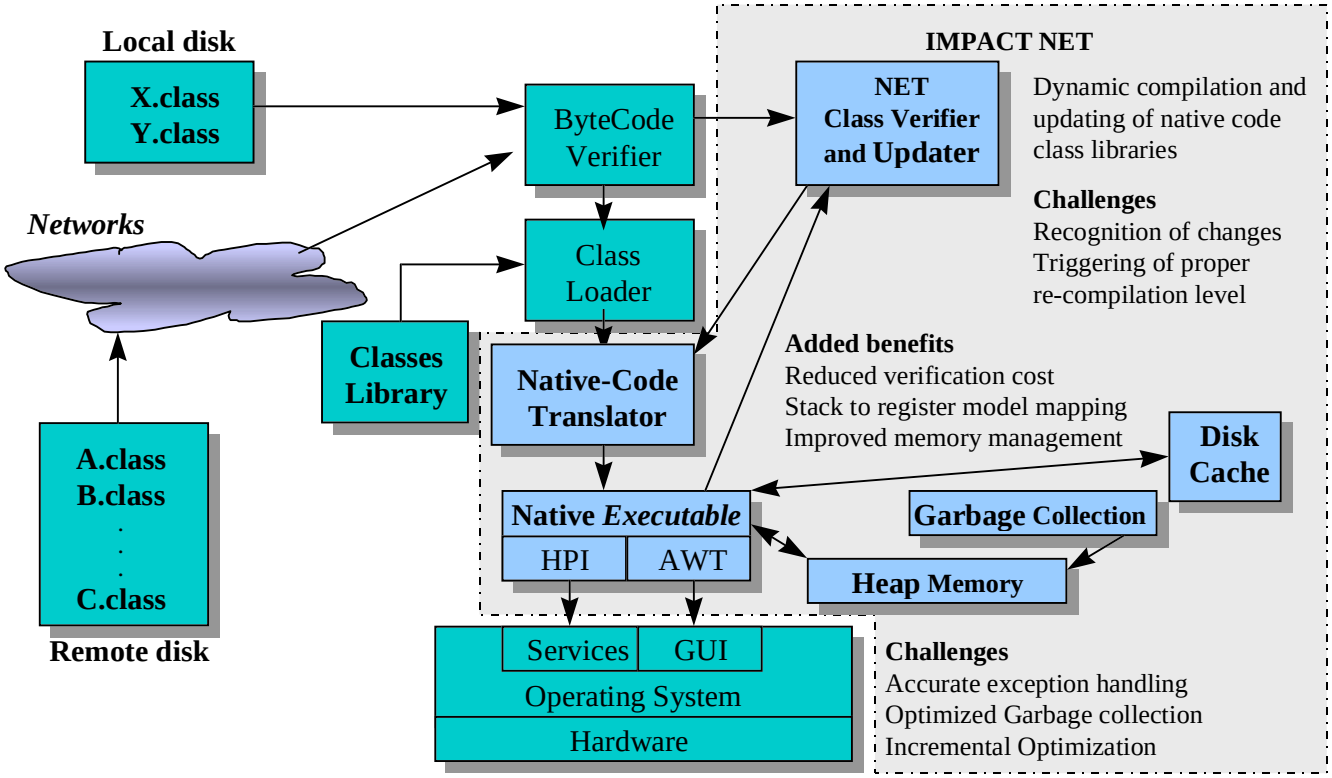

Figure 2: Overview of the IMPACT NET Compiler Framework

3.1 Taking Advantage of Architectural Registers

The Java VM uses a stack computation model where source operands are fetched from the top of the *operand stack*, and the result is pushed back onto this stack. The use of a stack model eliminates the need for making assumptions about the architectural register file size available to the interpreter, and thus increases the architectural independence of the interpreter[9]. However, the stack model requires that items be pushed and popped off a central stack, creating an execution bottleneck. For example, if we execute a code segment such as:

```
int A, B, C, j;
A = 4;
B = 8;
for(j = 0; j < 10; j++)
{ C = A + B;
 .... /* A and B are modified in the body of the loop */
}
```

a register-based architecture would load the values of A and B into a register, suffering the load expense only once per variable for the entire loop. In the Java VM, two numbers can only be added when both numbers are on top of the stack. Therefore, each time we compute C we must push A on the stack, push B on the stack, pop both of these values, add them, and push their result back on the stack. Even with the optimization of pushing both A and B at once, which is used in the Sun 1.0.2 VM[5], there is a substantial delay over the register execution model.

In the NET compiler approach, we improve upon the stack model by mapping the run-time stack to architectural registers. During this mapping it is important to utilize all the architectural registers in order to minimize both the number of native code instructions and the memory traffic. The first step in this register mapping translates a *push* to the operand stack into a *move* to a register, and a *pop* into a *move* from a register. Then optimizations, including register renaming, copy propagation, and dead code removal followed by global register allocation, eliminate all the overhead associated with the stack computation model. Further details on this mapping can be found in the paper by Hsieh, et. al[4].

3.2 Streamlining the Run-time Memory Organization

Java's ability to dynamically link a new class or interface at run-time, locating its fields as needed, overcomes the recompilation problems associated with class library updates in other object-oriented languages such as C++[1]. By dynamically locating the fields within the class, Java can ignore added fields, new methods, and new features as long as the original fields and structure remains valid.

Figure 3a illustrates the heap memory organization used by the SUN Java interpreter to facilitate dynamic linking. In this organization, neither the class object nor the array object points directly to its associated data. Rather, they each point to an 8-byte *handle*, which points to the corresponding data[10]. This indirection allows the VM to find the up-to-date data without recompiling after software updates. The handle is also used for garbage collection and heap

compaction. Accesses to both *class instance data* and the *array body* require two levels of indirection due to this handle space[9]. Accesses to the *method block* for method invocation are even more complex, and may need three or more levels of indirection.

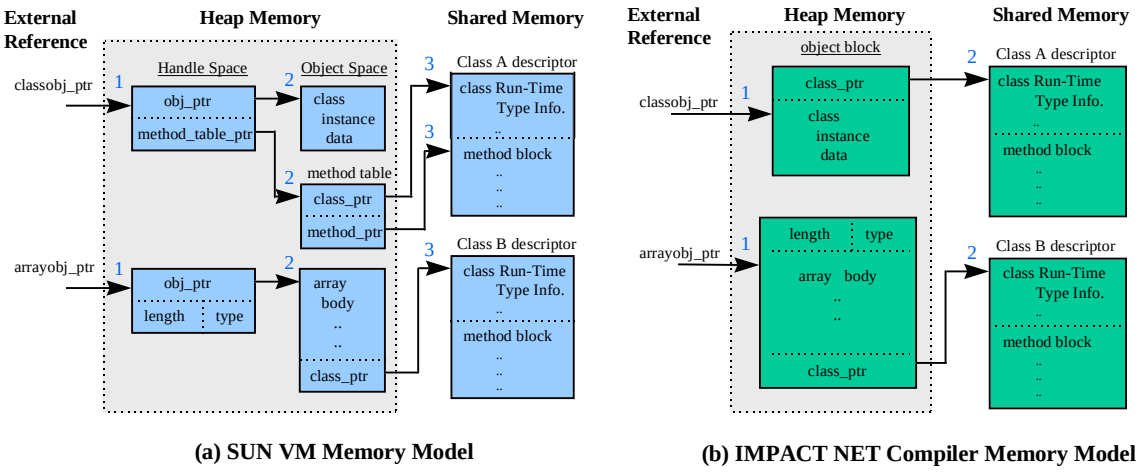

Figure 3: Memory Models

The enhanced memory model used in the our NET compiler (Figure 3b) reduces the amount of indirection by combining the *class instance data* block and the *method table* into one *object block*. A reference to our *object block* now requires only one level of indirection. Also, since the class run-time type information in our implementation remains constant at run time, we eliminate the *method_ptr* in SUN's Memory model shown in Figure 3a, reducing a method block reference to two levels of indirection. The complex method invocations are similarly optimized. An advantage of this approach is that it consumes less memory, and allows quicker access to data and class methods. The drawback is that during a software update, all references from other modules that invoke the updated module must be adjusted. This is handled by the updater shown in Figure 2.

3.3 Reducing the Time Spent Performing Garbage Collection

In C/C++ it is the responsibility of the programmer to handle dynamic memory allocation. Keeping track of all dynamically allocated memory in a large program is often complex. The Java language avoids problems that this complexity may cause by using garbage collection. Garbage collection transfers the responsibility for memory management from the programmer to the run-time system, or the VM implementation in Java's case. The job of the garbage collector (GC) is to periodically determine which memory blocks are still in use, and release the others for reuse by the memory manager. Java's GC runs as either a synchronous or asynchronous background thread, with the goal of interfering as little as possible with program execution. One use of the handle space in Sun's VM 1.0.2[5] is to facilitate the implementation of a GC. We investigated the effects of a baseline approach to the GC that does not rely on the handle space eliminated in our streamlining of the memory organization discussed in Section 3.2. We have implemented this GC in the IMPACT NET compiler, and compared it with the current GC used in Sun's VM 1.0.2.

Our baseline GC, shown in Figure 4, is modeled on what we call the "lazy approach" to garbage collection. Namely, we do not collect the garbage until we run out of room or reach some limit. To perform the actual garbage collection we use a mark and sweep approach. That is, we mark the memory blocks still in use and free the others. We accomplish our mark and sweep with the help of additional structures. Two of these structures are bit map arrays that hold the information necessary for collecting garbage as shown in Figure 4. In addition, we have organized memory so that all memory blocks contain a header field. The header fields for the memory blocks store information such as the size of the block. The size information is used when locating the position of the header for the next allocated memory block (Figure 4).

When the program is first started, three megabytes of memory is reserved for the memory manager and given an initial header. The next step is to initialize the Memory Bit Map (MBM). The MBM consists of single bit fields, with each bit corresponding to a consecutive 64 bits of

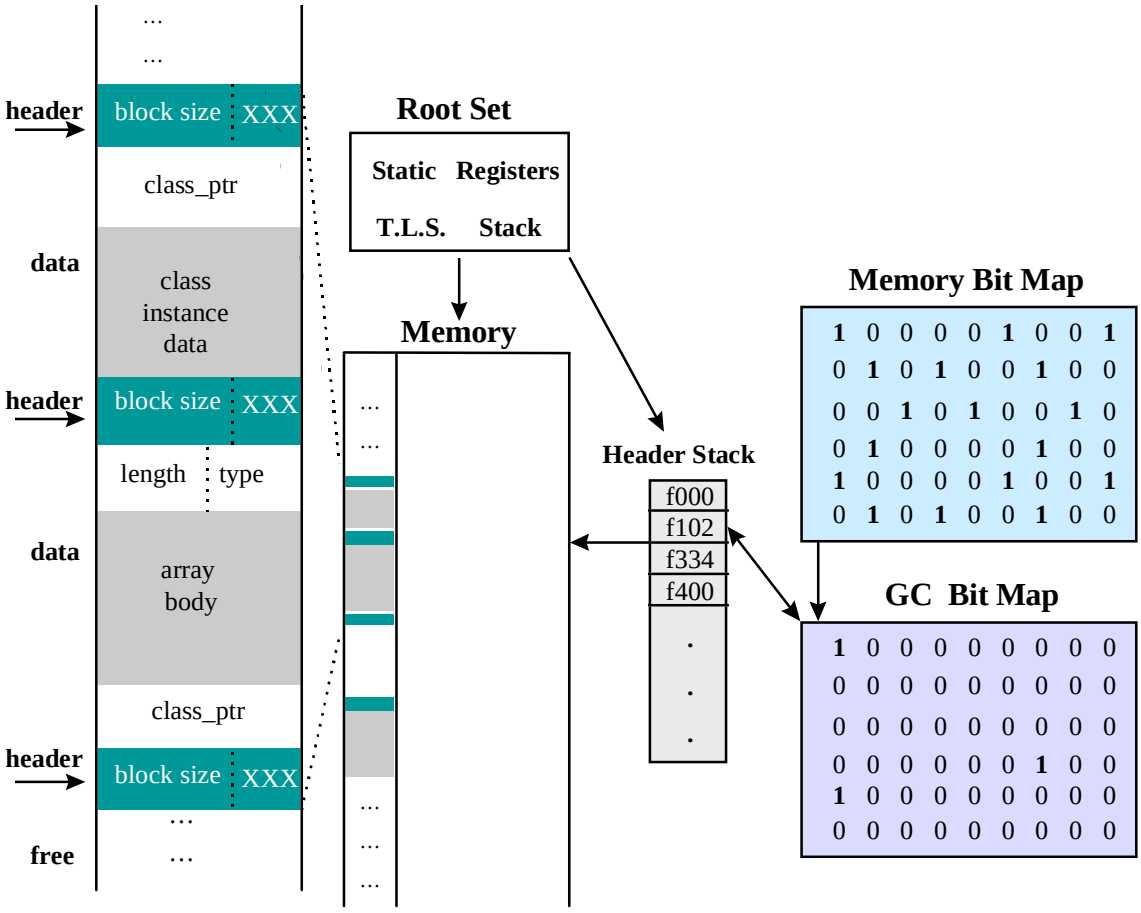

Figure 4: The IMPACT NET Compiler's Baseline Garbage Collector

memory. The MBM bits are used to record information concerning the allocated memory. At program startup, the MBM is initialized to all zeros, except for a leading one which corresponds to the initial header. As the program runs, the MBM is updated by setting the corresponding bit for the header of each newly allocated block of memory. Dynamically allocated objects are always referenced using the header address and an offset from the header.

The GC is called after the memory available to the memory manager has been consumed. It uses the Garbage Collector Bit Map (GCBM) to keep track of the memory blocks determined to

be still in use (*active* memory blocks). The GCBM is identical in size and shape to the MBM, and is initialized to all zeros. Its entries are changed to one as active memory blocks are found. The search for active memory blocks starts with the known active blocks contained in the root set. This root set includes the static memory blocks, registers in use, stack frame for method invocations, and thread local storage (TLS), as shown in Figure 4. To find the rest of the active blocks, each block known to be active is scanned for addresses of other memory blocks. Unfortunately, the GC cannot easily distinguish data representing memory block addresses from other types of data. In order to be conservative, the GC must treat every data value as a potential address. However, in order for an address to be valid it must correspond to the header of an allocated memory block. The GC can check this requirement efficiently, by determining if the corresponding MBM bit is set to one. When a valid address is found, the corresponding GCBM bit is set to one (if not already set) to indicate that this block must now be considered active. The first time a block is determined to be active, it is added to the header stack so that it also can be scanned for other possible active blocks, as shown in Figure 4. After every active block has been scanned, the GCBM contains a map of all active blocks and the free memory blocks can then be easily determined. To reduce memory fragmentation, consecutive free memory blocks are combined. When necessary, the memory manager can also allocate additional memory.

Our GC has shown some improvement over the model used by Sun. By using the lazy approach, the GC is not even called for smaller programs. We examined the execution time of garbage collection incurred by both the Sun interpreter and the NET compiler. Common C/C++ benchmarks that were hand-translated to Java source code, staying as close as possible to a one-to-one mapping, were used. For larger programs such as *javac* and *130.li*, the translated code GC activity is lower than that of Sun's interpreter. The time spent performing garbage collection in *130.li* was reduced from 2.76 to 0.98 seconds, and in *javac*, from 9.36 to 1.96 seconds.

3.4 Reducing the Verification Overhead

The key to security in Java is the verification process it uses. The Java bytecode verification is a four-pass process, three of which are conducted when the class is first loaded and linked and the fourth when the code is executed[2]. The first pass simply ensures that the loaded Java class file has the proper format. Although this pass can spot some initial problems within the class just loaded into the VM, it is not enough to ensure the full security needed in public network software distribution. The second and third passes in the Java VM verification scheme occur during the linking phase of the class loading. During the second pass the verifier checks for items dealing with language semantics and structure. The third pass of the verifier is an actual code verifier which examines the contents of the methods contained in the class. This pass actually performs a data flow analysis on each method as it checks the method. The fourth pass is conducted when the method is called (invoked), and includes both access permission and type checking.

In SUN's VM version 1.0.2, the first three passes in the verification process are only invoked for classes imported from another machine (untrusted source). However, due to the dynamic updating features, a remote class is imported every time a program accessing it is run, whether or not the class was updated since a program last accessed it. In the NET compiler, classes are validated during translation, leaving the native code on the local machine for future invocations of the program. This verification is performed again only when the updater spots a change in a class. In addition to the reduced verification overhead, we are also able to enforce a stricter security policy. For example, SUN's VM 1.0.2 does not verify classes that exist on the user's machine, allowing a user to install a program from another source and inadvertently bypass Java's security features. Every class is verified at translation time whether or not it is local.

3.5 Better Utilization of Processor Resources

The speed of executing programs in a modern system is not determined solely by the number of instructions executed. A significant amount of the execution time can be spent in dealing with inefficient use of the microarchitecture mechanisms such as the instruction and data caches. Translating Java bytecode into native code reduces the undesirable effects Java has on these architectural features, particularly when the VM is implemented via an interpreter.

The architectural performance studies were performed using the IMPACT simulation environment, which performs an execution-driven simulation, to generate results. Our cache simulation model consists of a 32Kbyte instruction cache (Icache) with 64-byte blocks and 2-way set-associativity. The data cache (Dcache) is 16Kbytes with 64-byte blocks and 2-way set-associativity.

3.5.1 Instruction Cache Performance

Due to the software emulation of the bytecode instructions in an interpreter, the processor is executing the interpreter program, rather than the benchmark application. The bytecode versions of the benchmarks, which are now input sets to the interpreter, have an impact on the size of the interpreter Icache footprint. Therefore, to examine Icache performance we looked for input set characteristics that affected the size of this footprint. As can be seen from the number of Icache misses in the third column of Table 1, the interpreter's Icache behavior is relatively consistent except when running *099.go* and *132.ijpeg*. These programs have a much larger unique bytecode mix in comparison with the other programs. In particular, *099.go* uses 123 of the 227 possible unique bytecode instructions, and *132.ijpeg* uses 141, while the others average around 100. Additionally, *099.go* and *132.ijpeg* utilize a larger number of complex bytecode instructions, such as array operations which require additional bounds checking overhead. Translation to native code will remove the interpreter effects from the Icache, however for large programs the Icache performance may not improve. For example, Icache misses for *wc* are significantly reduced after

TOTAL ICACHE MISSES			
BENCHMARKS	**C CODE**	**SUN INTERPRETER**	**NET COMPILER**
WC	13	6129	477
GREP	36	6251	484
026.COMPRESS	44	6726	545
099.GO	242114	18460	2873437
CMP	11	6124	450
DES	42	6680	1059
132.IJPEG	6402	50919	21294

Table 1: Icache performance of C code, Sun Interpreter, and the IMPACT NET compiler (32Kbyte Icache with 64-byte blocks and 2-way set-associativity).

translation to native code when compared with those of the interpreter, as shown in the fourth column of Table 1. On the other hand, the Icache misses of the translated code greatly increase for *099.go*, because it is a much larger program and has a larger Icache footprint than the interpreter. However, any Icache degradation due to the native code translation will be more then compensated for by the improved data cache performance, as will be discussed next in Section 3.5.2. Due to the extra instructions required for implementation of the Java specifications, such as array index bounds checking (AIBC), the translated code typically incurs more Icache misses than native C code, shown in column 2 of Table 1.

3.5.2 Data Cache Performance

As mentioned earlier, in the interpreter approach the Java bytecode effectively becomes data. In addition to the Java program's data and bytecode, all of the interpreter state is also competing for the limited Dcache space. The significant increase in Dcache traffic that results can be seen in the second column of Table 2, which shows the number of interpreter Dcache requests per native C code Dcache request. Although some of this overhead is due to fetching the bytecode, most is due to the interpreter's processing of the fetched bytecode.

As a result of the competition for Dcache space, there is also a dramatic increase in read misses, as shown in column three of Table 2. This effect is most noticeable for *wc*, *grep* and *cmp*,

which had particularly low numbers of misses in the native C code. On the other hand, the increases are smallest for *026.compress*, *099.go* and *132.ijpeg*, because they already have high numbers of misses in the native C code.

Translating the Java bytecode to native code separates the benchmark code from the data, as in native C code, and the extra interpreter state is not needed. The last two columns of Table 2 show that the number of Dcache read requests and misses incurred by the translated code, divided by the corresponding numbers for native C code, have been greatly reduced in comparison with the interpreter ratios. However, extra requests are still generated in order to support some of the Java specifications. For instance, before each array access, the array's size is loaded during array index bounds checking. Usually these extra accesses result in more misses in comparison with native C code, but in the case of *cmp*, the translated version actually incurs fewer misses. Further investigation revealed that one of the two input buffers was conflicting slightly with a global variable in the native C version, but that our translated version mapped the buffers differently, avoiding these conflict misses.

BENCHMARKS	SUN Interpreter		IMPACT NET Compiler	
	Read Requests Per C Read Request	Read Misses Per C Read Miss	Read Requests Per C Read Request	Read Misses Per C Read Miss
WC	214.6	9961.4	2.7	4.2
GREP	85.8	649.2	3.5	10.1
026.COMPRESS	78.5	3.5	2.6	1.0
099.GO	51.7	11.7	3.4	1.6
CMP	102.8	519.2	2.2	0.1
DES	75.6	172.3	3.6	8.4
132.IJPEG	70.4	26.1	3.9	2.4

Table 2: Dcache performance of Sun interpreter relative to C code (16Kbyte primary Dcache with 64-byte blocks and 2-way set-associativity).

4. Preliminary Comparison of Java Software Methods

As stated earlier, even though our NET compiler is still in the preliminary stages of development, we have already shown substantial performance improvement over other software methods for executing Java. We examined the performance of our NET compiler versus the Sun VM, Microsoft's JIT compiler, and native C/C++ code, on an 166MHz Pentium-based PC running Windows 95 with 48 Mbytes of memory. The Microsoft's JIT was chosen because it was the most stable, and able to handle the most benchmarks. There were only two benchmarks (*132.ijpeg* and *jBYTEmark*) that the Microsoft's JIT version 1.0 could not handle at the time of this evaluation, and the average performance numbers for the Microsoft's JIT does not include these benchmarks. The Java interpreter used was Sun JDK Version 1.0.2. The IMPACT NET compiler was used for the bytecode-to-native-code translation, and the C code was optimized for the x86 instruction set using the Microsoft Visual C++ compiler version 4.2.

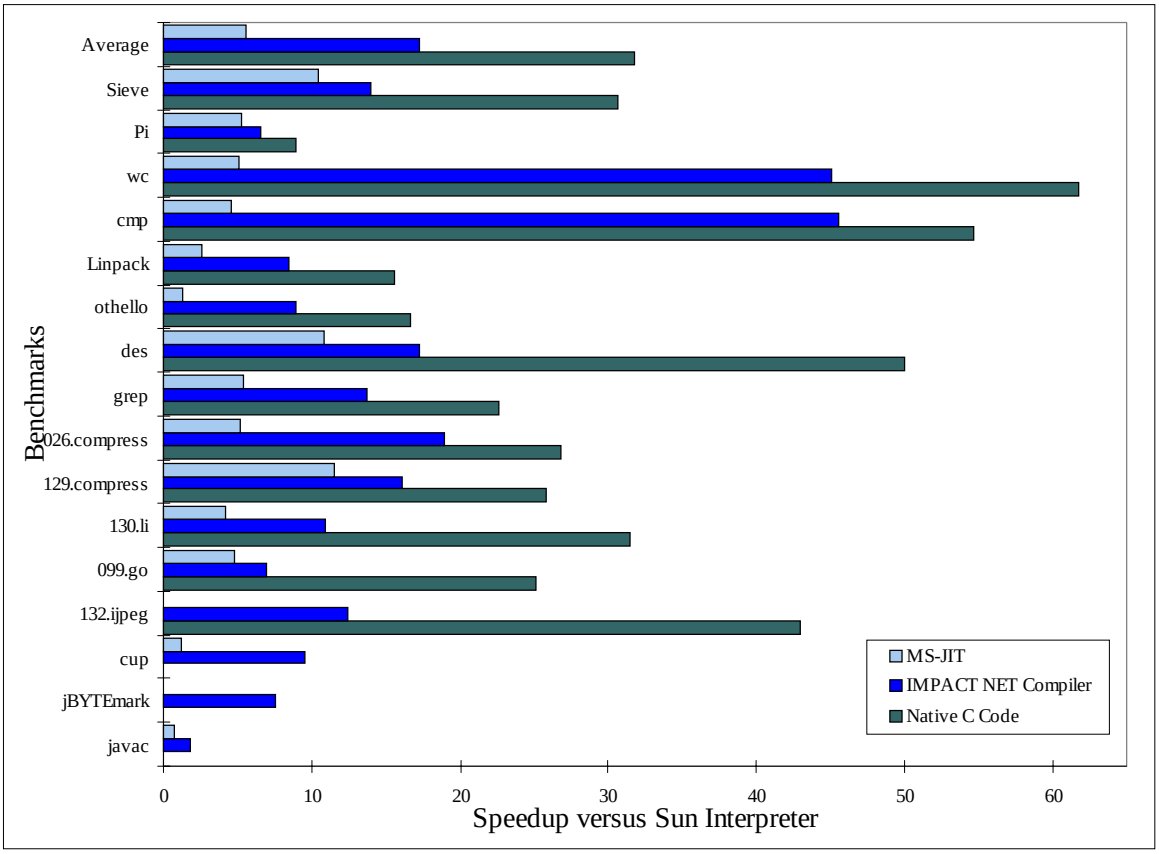

Figure 5: Performance Comparison of Software Methods for Executing Java Bytecode

Figure 5 shows the speedup of the different execution models over the Sun interpreter version 1.0.2. The IMPACT NET compiler is currently capturing 28-83% of the native C/C++ code performance, with speedups over the interpreter as high as a factor of 45.6. On average, for C/C++ benchmarks, the IMPACT NET compiler achieved 54% of native C/C++ performance, with an average factor of 17.3 speedup over the interpreter for all benchmarks. This compares to Microsoft's JIT's 23% of C/C++ performance, with an average factor of 5.6 speedup over the interpreter, and the SUN's interpreter's 4% of C/C++ performance. Although the initial performance captured by the IMPACT NET compiler is encouraging, we have identified several promising optimizations that we believe will significantly improve translated code performance.

One such optimization is the elimination of array index bounds checks that are found to be unnecessary through extensive program analysis[11]. There is a potential 15% average performance improvement if this overhead can be eliminated. When coupled with a class updater utilizing dynamic linking technology, we should be able to effectively utilize high-powered transformations that maintain the Java semantics while approaching C/C++ performance for a large set of applications.

Acknowledgments

The authors would like to thank all the members of the IMPACT research group whose comments and suggestions helped to improve the quality of this article, in particular Dan Lavery, Dan Connors, Dave August, Brian Deitrich and Rich Kutter. We would also like to thank the anonymous referees for their constructive comments.

This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948, 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).

References

1. J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification.* Addison-Wesley, Massachusetts. 1996.
2. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specifications.* Addison-Wesley, Massachusetts. 1997.
3. *Microsoft's Software Development Kit (SDK) for Java™,* November 1996. *http://www.microsoft.com/java/*
4. C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results," *Proceedings of the 29th International Symposium on Microarchitecture,* Paris, France, December 1996.
5. *Java™-Programming for the Internet,* Sun Microsystems, Inc., 1996, *http://java.sun.com/*
6. *Borland C++ Development Suite,* Borland International, Inc., 1996 , *http://www.borland.com/*
7. *Café – Visual Java Development and Debugging Tools*, Symantec Corporation, 1996, *http://www.symantec.com/*
8. 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, June 1991, pp. 266-275, *http://www.crhc.uiuc.edu/IMPACT/.*
9. K. Arnold and J. Gosling. *The Java™ Programming Language.* Addison-Wesley, Massachusetts. 1996.
10. F. Yellin and T. Lindholm, "Java™ Internals," *JavaOne Sun's Worldwide Java Developer Conference,* San Francisco, CA., May 29-31, 1996, *http://www.oasis.leo.org/java/documentation/slides/JavaOne/00-index.html*
11. Roger Alexander Bringmann, "Enhancing Instruction Level Parallelism Through Compiler-Controlled Speculation," Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, 1995, *http://www.crhc.uiuc.edu/IMPACT/.*

Cheng-Hsueh Andrew Hsieh is currently working as an intern at Intel in Santa Clara, CA. He is a Ph.D. candidate at the University of Illinois. He received his MS degree in electrical engineering in 1993 from University of Califormia at Los Angeles and his BS degree in electrical engineering in 1990 from the National Taiwan University. His research interests include universal software languages, compiler optimizations and binary translation. Hsieh has been a student member of IEEE since 1995.

Marie T. Conte is currently a Ph.D. candidate in the Department of Electrical Engineering at the University of Illinois. She received her BS degree in electrical engineering in 1995 from the University of Delaware. Her research interests include universal software languages, ubiquitous computing, compiler optimizations, and software translators. Conte has been a student member of IEEE since 1992 and a student member of ACM since 1994.

Teresa L. Johnson is currently a Ph.D. candidate in the Department of Electrical Engineering at the University of Illinois, where she received her MS in 1995 and her BS in 1993. Her research interests include cache memory optimizations for high-performance computer architectures, and universal software languages.

John C. Gyllenhaal is currently a Ph.D. candidate in the Department of Electrical Engineering at the University of Illinois, where he received his M.S. degree in 1994. He received his B.S. degree in electrical engineering at the University of Arizona in 1991. His research interests include schedule-time optimizations, universal software languages, and performance evaluation tools for high-performance

computer architectures.

Wen-mei W. Hwu is a Professor at the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research interest is in the area of architecture, implementation, and compilation for high performance computer systems. He is the director of the IMPACT project, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. In recognition of his contributions to the areas of compiler optimization and computer architecture, the Intel Corporation named him the Intel Associate Professor at the College of Engineering, University of Illinois in 1992. He received the National Eta Kappa Nu Outstanding Young Electrical Engineer Award for 1993, the 1994 Senior Xerox Award for Faculty Research, and the University Scholars Award of the University of Illinois. Dr. Hwu received his Ph.D. degree in Computer Science from the University of California, Berkeley, in 1987.

Article Summary

Java, the new object-oriented programming language from Sun Microsystems, appears to be setting the standard for universal software development. Java code compiles into a binary format called bytecode which can be used for software distribution, and which does not need recompilation in order to run on any platform. Java is also secure, guarding against code corruption before execution. This new language uses run-time resolution to locate objects and their corresponding classes, meaning software updates can be integrated as quickly as they are made available. However, there are several trade-offs and costs involved in migrating to a universal software distribution environment.

Currently there are four approaches to running Java: an interpreter, a Just-In-Time (JIT) compiler, a Native Executable Translation (NET) compiler, and a hardware implementation. In this article we present our initial prototyping experience with our NET compiler, an optimizing Java-bytecode-to-native-machine-code translator. 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/C++ programming languages. However, the work with our NET compiler is not limited to Java. Our goal is to develop a strong portfolio of techniques from our Java implementation efforts that will contribute to the development and acceptance of any universal software distribution language.

We focus our discussion on the critical issues involved in the design of our NET compiler, including minimizing verification overhead, mapping the stack computation model of the bytecode virtual machine to the register computation model of modern processors and developing a more efficient memory organization. We also explain some of the less intuitive costs involved in running Java. Preliminary results show that our optimizing NET compiler is currently capable of achieving better performance than the other bytecode execution methods, in some cases achieving speeds comparable to directly compiled native code.