Techniques for Obtaining High Performance in Java Programs

IFFAT H. KAZI, HOWARD H. CHEN, BERDENIA STANLEY, AND DAVID J. LILJA

Minnesota Supercomputing Institute, University of Minnesota

This survey describes research directions in techniques to improve the performance of programs written in the Java programming language. The standard technique for Java execution is interpretation, which provides for extensive portability of programs. A Java interpreter dynamically executes Java bytecodes, which comprise the instruction set of the Java Virtual Machine (JVM). Execution time performance of Java programs can be improved through compilation, possibly at the expense of portability. Various types of Java compilers have been proposed, including Just-In-Time (JIT) compilers that compile bytecodes into native processor instructions on the fly; direct compilers that directly translate the Java source code into the target processor's native language; and bytecode-to-source translators that generate either native code or an intermediate language, such as C, from the bytecodes. Additional techniques, including bytecode optimization, dynamic compilation, and executing Java programs in parallel, attempt to improve Java run-time performance while maintaining Java's portability. Another alternative for executing Java programs is a Java processor that implements the JVM directly in hardware. In this survey, we discuss the basic features, and the advantages and disadvantages, of the various Java execution techniques. We also discuss the various Java benchmarks that are being used by the Java community for performance evaluation of the different techniques. Finally, we conclude with a comparison of the performance of the alternative Java execution techniques based on reported results.

Categories and Subject Descriptors: A.1 [**General Literature**]: Introductory and Survey; C.4 [**Computer Systems Organization**]: Performance of Systems; D.3 [**Software**]: Programming Languages

General Terms: Languages, Performance

Additional Key Words and Phrases: Java, Java virtual machine, interpreters, just-in-time compilers, direct compilers, bytecode-to-source translators, dynamic compilation

1. INTRODUCTION

The Java programming language that evolved out of a research project started by Sun Microsystems in 1990 [Arnold and Gosling 1996; Gosling et al. 1996] is one of the most exciting technical developments in recent years. Java combines several features found in different programming paradigms into one language. Features such as platform independence for portability, an object-orientation model, support for multithreading, support for distributed programming, and automatic garbage collection, make Java very appealing to program developers. Java's

Authors' addresses: I. H. Kazi, Dept. of Electrical and Computer Engineering, Minnesota Supercomputing Inst., Univ. of Minnesota, 200 Union St., SE, Minneapolis, MN 55455; H. Chen, Dept. of Computer Science and Engineering; B. Stanley, Dept. of Electrical and Computer Engineering; D. J. Lilja, Dept. of Electrical and Computer Engineering.

"write-once, run anywhere" philosophy captures much of what developers have been looking for in a programming language in terms of application portability, robustness, and security. The cost of Java's flexibility, however, is its slow performance due to the high degree of hardware abstraction it offers.

To support portability, Java source code is translated into architecture neutral bytecodes that can be executed on any platform that supports an implementation of the Java Virtual Machine (JVM). Most JVM implementations execute Java bytecodes through either interpretation or Just-In-Time (JIT) compilation. Since both interpretation and JIT compilation require runtime translation of bytecodes, they both result in relatively slow execution times for an application program. While advances with JIT compilers are making progress towards improving Java performance, existing Java execution techniques do not yet match the performance attained by conventional compiled languages. Of course, performance improves when Java is compiled directly to native machine code, but at the expense of diminished portability.

This survey describes the different execution techniques that are currently being used with the Java programming language. Section 2 describes the basic concepts behind the JVM. Section 3 discusses the different Java execution techniques, including interpreters, JIT and static compilers, and Java processors. In Section 4, we describe several optimization techniques for improving Java performance, including dynamic compilation, bytecode optimization, and parallel and distributed techniques. Section 5 reviews the existing benchmarks available to evaluate the performance of the various Java execution techniques with a summary of their performance presented in Section 6. Conclusions are presented in Section 7.

2. BASIC JAVA EXECUTION

Basic execution of an application written in the Java programming language begins with the Java source code. The Java source code files (*.java* files) are translated by a Java compiler into Java *bytecodes*, which are then placed into *.class* files. The bytecodes define the instruction set for the JVM which actually executes the user's application program.

2.1. Java Virtual Machine

The *Java Virtual Machine* (JVM) executes the Java program's bytecodes [Lindholm and Yellin 1997; Meyer and Downing 1997]. The JVM is said to be *virtual* since, in general, it is implemented in software on an existing hardware platform. The JVM must be implemented on the target platform before any compiled Java programs can be executed on that platform. The ability to implement the JVM on various platforms is what makes Java portable. The JVM provides the interface between compiled Java programs and any target hardware platform.

Traditionally, the JVM executes the Java bytecodes by interpreting a stream of bytecodes as a sequence of instructions. One stream of bytecodes exists for each method[1] in the class. They are interpreted and executed when a method is invoked during the execution of the program. Each of the JVM's stack-based instructions consists of a one-byte *opcode* immediately followed by zero or more *operands*. The instructions operate on *byte*, *short*, *integer*, *long*, *float*, *double*, *char*, *object*, and *return address* data types. The JVM's instruction set defines 200 standard opcodes, 25 *quick variations* of some opcodes (to support efficient dynamic binding), and three reserved opcodes. The opcodes dictate to the JVM what action to perform. Operands provide additional information, if needed, for the JVM to execute the action. Since bytecode instructions operate primarily on a stack, all operands must be pushed on the stack before they can be used.

The JVM can be divided into the five basic components shown in Figure 1. Each of the *registers*, *stack*, *garbage-collected heap*, *methods area*, and *execution engine*

[1] A *method* roughly corresponds to a function call in a procedural language.

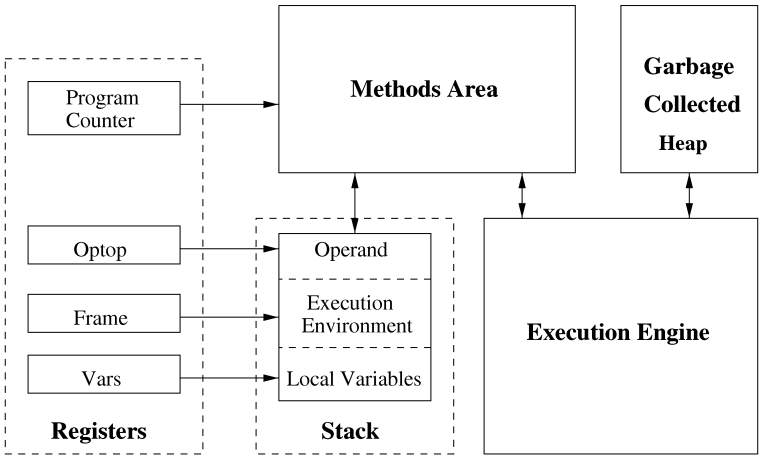

Fig. 1. Basic components of the Java Virtual Machine.

components must be implemented in some form in every JVM. The *registers* component includes a *program counter* and three other registers used to manage the stack. Since most of the bytecode instructions operate on the stack, only a few registers are needed. The bytecodes are stored in the *methods area*. The *program counter* points to the next byte in the *methods area* to be executed by the JVM. Parameters for bytecode instructions, as well as results from the execution of bytecode instructions, are stored in the *stack*. The stack passes parameters and return values to and from the methods. It is also used to maintain the state of each method invocation, which is referred to as the *stack frame*. The *optop*, *frame*, and *vars* registers manage the stack frame.

Since the JVM is a stack-based machine, all operations on data must occur through the *stack*. Data is pushed onto the *stack* from constant pools stored in the methods area and from the local variables section of the stack. The *stack frame* is divided into three sections. The first is the *local variables* section which contains all of the local variables being utilized by the current method invocation. The *vars* register points to this section of the stack frame. The second section of the stack frame is the *execution environment*, which maintains the stack operations. The *frame* register points to this section. The final section is the *operand stack*. This section is utilized by the bytecode instructions for storing parameters and temporary data for expression evaluations.

The *optop* register points to the top of the operand stack. It should be noted that the *operand stack* is always the topmost stack section. Therefore, the *optop* register always points to the top of the entire stack. While instructions obtain their operands from the top of the stack, the JVM requires random access into the stack to support instructions like *iload*, which loads an integer from the local variables section onto the operand stack, or *istore*, which pops an integer from the top of the operand stack and stores it in the local variables section.

Memory is dynamically allocated to executing programs from the *garbage-collected heap* using the *new* operator. The JVM specification requires that any space allocated for a new object be preinitialized to zeroes. Java does not permit the user to explicitly free allocated memory. Instead, the garbage collection process monitors existing objects on the heap and periodically marks those that are no longer being used by the currently executing Java program. Marked objects are then returned to the pool of available memory. Implementation details of the garbage collection mechanism are discussed further in Section 2.2.

The core of the JVM is the *execution engine*, which is a "virtual" processor

that executes the bytecodes of the Java methods. This "virtual" processor can be implemented as an interpreter, a compiler, or a Java-specific processor. Interpreters and compilers are software implementations of the JVM while Java processors implement the JVM directly in hardware. The execution engine interacts with the methods area to retrieve the bytecodes for execution. Various implementations of the Java execution engine are described in subsequent sections.

2.2. Garbage Collection

In Java, objects are never explicitly deleted. Instead, Java relies on some form of garbage collection to free memory when objects are no longer in use. The JVM specification [Lindholm and Yellin 1997] requires that every Java runtime implementation should have some form of automatic garbage collection. The specific details of the mechanism are left up to the implementors. A large variety of garbage collection algorithms have been developed, including *reference counting*, *mark-sweep*, *mark-compact*, *copying*, and *noncopying implicit collection* [Wilson 1992]. While these techniques typically halt processing of the application program when garbage collection is needed, *incremental* garbage collection techniques have been developed that allow garbage collection to be interleaved with normal program execution. Another class of techniques known as *generational* garbage collection improve efficiency and memory locality by working on a smaller area of memory. These techniques exploit the observation that recently allocated objects are most likely to become garbage within a short period of time.

The garbage collection process imposes a time penalty on the user program. Consequently, it is important that the garbage collector is efficient and interferes with program execution as little as possible. From the implementor's point of view, the programming effort required to implement the garbage collector is another consideration. However, easy-to-implement techniques may not be the most execution-time efficient. For example, conservative garbage collectors treat every register and word of allocated memory as a potential pointer and thus do not require any additional type information for allocated memory blocks to be maintained. The drawback, however, is slower execution time. Thus, there are trade-offs between ease of implementation and execution-time performance to be made when selecting a garbage collection technique for the JVM implementation.

3. ALTERNATIVE EXECUTION TECHNIQUES FOR JAVA PROGRAMS

In addition to the standard interpreted JVM implementation, a variety of execution techniques have been proposed to reduce the execution time of Java programs. In this section, we discuss the alternative execution techniques summarized in Figure 2.

As shown in this figure, there are numerous alternatives for executing Java programs compared to the execution of programs written in a typical programming language such as C. The standard mechanism for executing Java programs is through interpretation, which is discussed in Section 3.1. Compilation is another alternative for executing Java programs and a variety of Java compilers are available that operate on either Java source code or Java bytecodes. We describe several different Java compilers in Section 3.2. Finally, Java processors, which are hardware implementations of the JVM, are discussed in Section 3.3.

3.1. Java Interpreters

Java interpreters are the original method for executing Java bytecodes. An interpreter emulates the operation of a processor by executing a program, in this case, the JVM, on a target processor. In other words, the running JVM program reads and executes each of the bytecodes of the user's application program in order. An interpreter has several advantages over a traditional compiled execution. Interpretation is very simple,

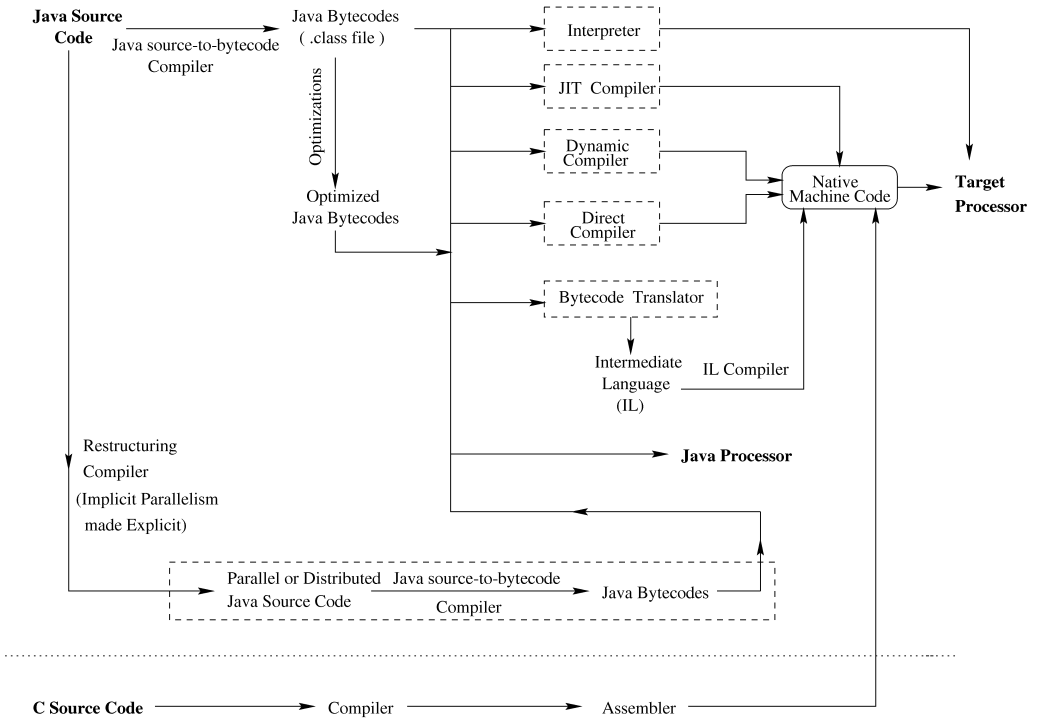

Fig. 2. Alternatives for executing Java compared to a typical C programming language compilation process.

and it does not require a large memory to store the compiled program. Furthermore, interpreters are relatively easy to implement. However, the primary disadvantage of an interpreter is its slow performance.

There are several existing Java interpreters. The Sun Java Developers Kit (JDK) [Sun Microsystems] is used to develop Java applications and applets that will run in all Java-enabled web browsers. The JDK contains all of the necessary classes, source files, applet viewer, debugger, compiler, and interpreter. Versions exist that execute on the SPARC Solaris, x86 Solaris, Windows NT, Windows 95, and Macintosh platforms. The Sun JVM is itself implemented in the C programming language. The Microsoft Software Development Kit (SDK) also provides the tools to compile, execute, and test Java applets and applications [Microsoft SDK Tools]. The SDK contains the Microsoft Win32 Virtual Machine for Java (Microsoft VM), classes, APIs, and so forth for the x86 and ALPHA platforms.

3.2. Java Compilers

Another technique to execute Java programs is with a compiler that translates the Java bytecodes into native machine code. Like traditional high-level language compilers, a *direct Java compiler* starts with an application's Java source code (or, alternatively, with its bytecode) and translates it directly into the machine language of the target processor. The *JIT compilers*, on the other hand, are dynamically-invoked compilers that compile the Java bytecode during runtime. These compilers can apply different optimizations to speed up the execution of the generated native code. We discuss some common Java-specific compiler optimizations in Section 3.2.1. The subsequent sections describe the various Java compilers and their specific features.

3.2.1. Java Compiler Optimizations

Due to the unique features of the JVM, such as the stack architecture, dynamic loading, and exception handling, Java native code compilers need to apply different types of optimizations than those used in traditional compilers. Some commonly implemented optimizations include mapping stack variables to machine registers, moving constant stack variables into native instructions, increasing the efficiency of exception handling, and inlining of methods.

Mapping variables to machine registers increases native code performance since registers are much faster to access than memory. Registers can also be used to reduce the amount of stack duplication in Java. Specifically, each Java bytecode instruction consumes stack variables. Therefore, if a bytecode operand is used by multiple bytecode instructions, it must be duplicated on the stack. Register operands can be used over multiple instructions, however, eliminating the stack duplication overhead for those instructions.

Stack activity also can be reduced by moving constants on the stack into native instructions as immediate operands. Java bytecode instructions must receive constant operands from the stack. Since most architectures support instructions with immediate operands, this optimization eliminates the overhead of loading the constant into a register or memory location.

Exception overhead can be reduced by eliminating unnecessary exception checks and increasing the efficiency of the exception checking mechanism. For example, arrays are often used in iterative loops. If an array index value remains bounded inside of a loop, array bounds checks can be eliminated inside of the loop, which can produce significant performance advantages for large arrays.

Inlining of static methods is useful for speeding up the execution of method calls. However, the Java to bytecode compiler can only inline static methods within the class because static methods in other classes may be changed before actual execution. The implementation of any single class instantiation is stable at runtime and can therefore be inlined. The JIT compilers described in Section 3.2.2 can make use of this fact to inline small static methods to thereby reduce the overall number of static method calls. Dynamic methods can be converted to static methods given a set of classes using class hierarchy analysis [Dean et al. 1995]. If a virtual method has not been overloaded in the class hierarchy, dynamic calls to that method can be replaced by static calls. However, class hierarchy analysis may be invalidated if a new class is dynamically loaded into the program.

3.2.2. Just-In-Time Compilers

A *Just-In-Time* (JIT) compiler translates Java bytecodes into equivalent native machine instructions as shown in Figure 2. This translation is performed at runtime immediately after a method is invoked. Instead of interpreting the code for an invoked method, the JIT compiler translates a method's bytecodes into a sequence of native machine instructions. These native instructions are executed in place of the bytecodes. Translated bytecodes are then cached to eliminate redundant translation of bytecodes.

In most cases, executing JIT compiler generated native code is more efficient than interpreting the equivalent bytecodes since an interpreter identifies and interprets a bytecode every time it is encountered during execution. JIT compilation, on the other hand, identifies and translates each instruction only once—the first time a method is invoked. In programs with large loops or recursive methods, the combination of the JIT compilation and native code execution times can be drastically reduced compared to an interpreter. Additionally, a JIT compiler can speed up native code execution by optimizing the code it generates, as described in Section 3.2.1.

Since the total execution time of a Java program is a combination of compilation time and execution time, a JIT compiler needs to balance the time spent optimizing

generated code against the time it saves by the optimization. Code optimization is further limited by the scope of compilation. Since compilation occurs on demand for one class or one method at a time, it is difficult for a JIT compiler to perform nonlocal optimizations. Due to the time and scope restrictions of JIT compilation, JIT optimizations are generally simple ones that are expected to yield reasonably large performance gains compared to the optimization time.

Although JIT compilers are generally more efficient than interpreters, there are still advantages to using an interpreter. One advantage is that interpreters are better suited to debugging programs. Another advantage is that JIT compilation compiles an entire method at once, while interpretation translates only the instructions that actually are executed. If only a small percentage of the bytecodes in a method are ever executed and the method is rarely executed, the time spent on JIT compilation may never be recouped by the reduction in execution time.

Current JIT compilers offer a variety of target platforms and features. The *Symantec Cafe JIT* is included in the Java 2 runtime environment for Windows 95/NT and Netscape Navigator [Symantec]. *Microsoft* includes a JIT with Internet Explorer for Windows 95/NT and Macintosh [Just-In-Time Compilation]. No published information is available about the implementation details of these JIT compilers.

IBM includes an optimizing JIT compiler in its IBM Developer Kit for Windows [Ishizaki et al. 1999; Suganuma et al. 2000]. This JIT compiler performs a variety of optimizations including method inlining, exception check elimination, common subexpression elimination, loop versioning, and code scheduling. The compiler begins by performing flow analysis to identify basic blocks and loop structure information for later optimizations. The bytecodes are transformed into an internal representation, called *extended bytecodes*, on which the optimizations are performed.

The optimizations begin with method inlining. Empty method calls originating from object constructors or small access methods are always inlined. To avoid code expansion (and thus, the resulting cache inefficiency) other method calls are inlined only if they are in program hotspots such as loops. Virtual method calls are handled by adding an explicit check to make sure that the inlined method is still valid. If the check fails, standard virtual method invocation is performed. If the referenced method changes frequently, the additional check and inlined code space adds additional overhead to the invocation. However, the referenced method is likely to remain the same in most instances, eliminating the cost of a virtual method lookup and invocation.

Following inlining, the IBM JIT compiler performs general exception check elimination and common subexpression elimination based on program flow information. The number of array bound exception checks is further reduced using *loop versioning*. Loop versioning creates two versions of a target loop—a safe version with exception checking and an unsafe version without exception checking. Depending on the loop index range test at the entry point of the loop, either the safe or the unsafe version of the loop is executed.

At this point, the IBM JIT compiler generates native x86 machine code based on the extended bytecode representation. Certain stack manipulation semantics are detected in the bytecode by matching bytecode sequences known to represent specific stack operations. Register allocation is applied by assigning registers to stack variables first and then to local variables based on usage counts. Register allocation and code generation are performed in the same pass to reduce compilation time. Finally, the generated native code is scheduled within the basic block level to fit the requirements of the underlying machine.

Intel includes a JIT compiler with the VTune optimization package for Java that interfaces with the Microsoft JVM [Adl-Tabatabai et al. 1998]. The Intel JIT compiler performs optimizations and generates code in a single pass without

generating a complete internal representation of the program. This approach speeds up native code generation while limiting the scope of the optimizations to extended basic blocks only. The Intel JIT compiler applies common subexpression elimination (CSE) within basic blocks, local and global register allocation, and limited exception optimizations.

The compiler first performs a linear scan of the bytecodes to record stack and variable information for use in later stages. This is followed by a global register allocation, code generation, code emission, and code patching pass. The compiler uses two alternative schemes for global register allocation. The first scheme allocates the 4 callee-saved registers to the variables with the highest static reference counts for the duration of the method. The second scheme iterates through all of the variables in order of decreasing static reference counts, allocating a register to a variable if the register is available in all of the basic blocks that the variable is referenced.

CSE is implemented by matching nonoverlapping subsequences in expressions. Basic blocks are represented as a string of bytes where common subexpressions are detected as duplicate substrings in the string. Any two matched substrings of less than sixteen bytecodes is considered as a candidate for elimination. If the value calculated during the first expression instance is still available during the second instance of the expression, the previous value is reused and the duplicate expression is eliminated. In this scheme, transitivity of expressions is not considered. Therefore, the expression "$x + y$" would not match with "$y + x$."

Unnecessary array bounds checks are detected by keeping track of the maximum constant array index that is bounds checked and eliminating index bound checks for constant values less than the maximum constant array index check. For example, if array location 10 is accessed before location 5, the bounds check for array location 5 is eliminated. This optimization is useful during array initialization since the array creation size is considered a successful bounds check of the highest array index. This eliminates bound checks for any initialization of array values using constant array indices. However, elimination of bounds checks does not apply to variables nor does it extend beyond the scope of a basic block.

Code for handling thrown exceptions is moved to the end of the generated code for a method. By placing exception handling code at the end of the method, the static branch predictor on Pentium processors will predict the branches to be not taken and the exception handling code is less likely to be loaded into a cache line. Since exceptions do not occur frequently, this is well-suited for the common case execution.

The OpenJIT project [Matsuoka et al. 1998] is a reflective JIT compiler written in Java. The compiler is reflective in the sense that it contains a set of self-descriptive modules that allow a user program to examine the internal state of the compiler's compilation and modify the state through a compiler-supplied interface. This interface allows user programs to perform program-specific optimizations by defining program-specific semantics. For instance, defining properties of commutivity and transitivity for a user object provides the compiler with more information to perform useful optimizations.

The open-source Kaffe project provides a JIT compiler for the Kaffe JVM that supports various operating systems on the x86, Sparc, M68k, MIPS, Alpha, and PARisc architectures [Wilkinson, Kaffe v0.10.0]. The Kaffe JIT compiler uses a machine-independent front-end that converts bytecodes to an intermediate representation called the KaffeIR. The KaffeIR is then translated using a set of macros which define how KaffeIR instructions map to native code.

The *AJIT compilation system* generates annotations in bytecode files to aid the JIT compilation [Azevedo et al. 1999]. A Java to bytecode compiler generates annotations that are stored as additional code attributes in generated class files to maintain compatibility with existing JVMs. These annotations carry compiler optimization-related information that

allow the JIT compiler to generate optimized native code without extensive runtime analysis. An example of a generated attribute in the AJIT system is the mapping of variables to an infinite virtual register set. This virtual register allocation (VRA) annotation is used by an annotation-reading JIT compiler to speed up register allocation and to identify unnecessary duplication of stack variables.

CACAO is a stand-alone JIT compiler for the DEC ALPHA architecture [Krall and Grafl 1997]. The CACAO compiler translates bytecodes to an intermediate representation, performs register allocation, and replaces constant operands on the stack with immediate instruction operands.

Fajita [FAJITA] is a variation of a JIT compiler that runs as a Java compilation server on a network, independent of the Java runtime. The server compiles Java bytecode as a pass-through proxy server, allowing class compilation to be cached and shared among different programs and machines. Since the lifetime of compiled code in this environment is generally much longer than a standard JIT, this compiler has more time to perform advanced optimizations.

3.2.3. Direct Compilers

A *direct compiler* translates either Java source code or bytecodes into machine instructions that are directly executable on the designated target processor (refer to Figure 2). The main difference between a direct compiler and a JIT compiler is that the compiled code generated by a direct compiler is available for future executions of the Java program. In JIT compilation, since the translation is done at run-time, compilation speed requirements limit the type and range of code optimizations that can be applied. Direct compilation is done statically, however, and so can apply traditional time-consuming optimization techniques, such as data-flow analysis, interprocedural analysis, and so on [Aho et al. 1986], to improve the performance of the compiled code. However, because they provide static compilation, direct compilers cannot support dynamic class loading. Direct compilation also results in a loss of portability since the generated code can be executed only on the specific target processor. It should be noted that the portability is not completely lost if the original bytecodes or source code are still available, however.

Caffeine [Hsieh et al. 1996] is a Java bytecode-to-native code compiler that generates optimized machine code for the X86 architecture. The compilation process involves several translation steps. First, it translates the bytecodes into an internal language representation, called *Java IR*, that is organized into functions and basic blocks. The Java IR is then converted to a machine-independent IR, called *Lcode*, using stack analysis, stack-to-register mapping, and class hierarchy analysis. The IMPACT compiler [Chang et al. 1991] is then used to generate an optimized machine-independent IR by applying optimizations such as inlining, data-dependence and interclass analysis to improve instruction-level parallelism (ILP). Further, machine-specific optimizations, including peephole optimization, instruction scheduling, speculation, and register allocation, are applied to generate the optimized machine-specific IR. Finally, optimized machine code is generated from this machine-specific IR. Caffeine uses an enhanced memory model to reduce the overhead due to additional indirections specified in the standard Java memory model. It combines the Java class instance data block and the method table into one object block and thus requires only one level of indirection. Caffeine supports exception handling only through array bounds checking. It does not support garbage collection, threads, and the use of graphics libraries.

The *Native Executable Translation* (NET) compiler [Hsieh et al. 1997] extends the Caffeine prototype by supporting garbage collection. It uses a mark-and-sweep garbage collector which is invoked only when memory is full or reaches a predefined limit. Thus, NET eliminates the overhead due to garbage collection in smaller application programs that have

low memory requirements. NET also supports threads and graphic libraries.

The IBM *High Performance Compiler for Java* (HPCJ) [Seshadri 1997] is another optimizing native code compiler targeted for the AIX, OS/2, Windows95, and WindowsNT platforms. HPCJ takes both Java source code and bytecodes as its input. If the Java source code is used as an input, it invokes the AIX JDK's Java source-to-bytecode compiler (javac) to produce the bytecodes. Java bytecodes are translated to an internal compiler intermediate language (IL) representation. The common back-end from IBM's XL family of compilers for the RS/6000 is used to translate the IL code into an object module (.o file). The object module is then linked to other object modules from the Java application program and libraries to produce the executable machine code. The libraries in HPCJ implement garbage collection, Java APIs, and various runtime system routines to support object creation, threads, and exception handling.

A common back-end for the native code generation allows HPCJ to apply various language-independent optimizations, such as instruction scheduling, common subexpression elimination, intramodular inlining, constant propagation, global register allocation, and so on. The bytecode-to-IL translator reduces the overhead for Java run-time checking by performing a simple bytecode simulation during basic block compilation to determine whether such checks can be eliminated. HPCJ reduces the overhead due to method indirection by emitting direct calls for instance methods that are known to be final or that are known to belong to a final class. It uses a conservative garbage collector since the common back-end does not provide any special support for garbage collection.

3.2.4. Bytecode-to-Source Translators

Bytecode-to-source translators are static compilers that generate an intermediate high-level language, such as C, from the Java bytecodes (see Figure 2). A standard compiler for the intermediate language is then used to generate executable machine code. Choosing a high-level language such as C as an intermediate language allows the use of existing compiler technology, which is useful since many compilers that incorporate extensive optimization techniques are available for almost all platforms.

Toba [Proebsting et al. 1997] is a bytecode-to-source translator for Irix, Solaris, and Linux platforms that converts Java class files directly into C code. The generated C code can then be compiled into machine code. Toba describes itself as a Way-Ahead-of-Time compiler since it compiles bytecodes before program execution in contrast to JIT compilers which compile the bytecodes immediately before execution. All versions of Toba provide support for garbage collection, exceptions, threads, and the BISS-AWT, an alternative to the Sun AWT (Abstract Window Toolkit). In addition, the Linux version of Toba also supports dynamic loading of classes using a JIT compiler. The Toba C code generator naively converts each bytecode directly into equivalent C statements without a complex intermediate representation, relying on the C compiler to eventually optimize the code.

Harissa [Muller et al. 1997, Welcome to Harissa] is a Java environment that includes both a bytecode translator and an interpreter for SunOS, Solaris, Linux, and DEC Alpha platforms. The Harissa compiler reads in Java source code and converts it into an intermediate representation (IR). It then analyzes and optimizes the structure of the Java code and outputs optimized C files. Since existing C compilers can be used to perform more generalized optimizations, Harissa's compiler focuses on IR optimizations such as static evaluation to eliminate the stack, elimination of redundant bounds checks, elimination of bounds checks on array indices that can be statically determined, and transforming virtual method calls into simple procedure calls using class hierarchy analysis. A complete interpreting JVM has been integrated into the runtime library to allow code to be dynamically loaded into previously compiled

applications. Since data structures are compatible between the compiled code and the interpreter, Harissa provides an environment that cleanly allows the mixing of bytecodes and compiled code.

TurboJ is a Java bytecode-to-source translator [TurboJ Java to Native Compiler] that also uses C as an intermediate representation. The generated code retains all of the Java run-time checks, although redundant checks are eliminated. TurboJ is not a stand-alone Java runtime system. Instead it operates in conjunction with a Java runtime system and uses the native JDK on a given platform for memory allocation, garbage collection, and access to the standard thread package.

Vortex is an optimizing compiler that supports general object-oriented languages [UW Cecil/Vortex Project, Dean et al. 1996]. The Java, C++, Cecil, and Modula-3 languages are all translated into a common IL. If the Java source code is not available, this system can use a modified version of the *javap* bytecode disassembler to translate the bytecodes into the Vortex IL. The IL representation of the program is then optimized using such techniques as intraprocedural class analysis, class hierarchy analysis, receiver class prediction, and inlining. An enhanced CSE technique is also provided. Once optimization is complete, either C code or assembly code can be generated.

3.3. Java Processors

To run Java applications on general-purpose processors, the compiled bytecodes need to be executed through an interpreter or through some sort of compilation, as described in the previous sections. While a JIT compiler can provide significant speedups over an interpreter, it introduces additional compilation time and can require a large amount of memory. If the bytecodes can be directly executed on a processor, the memory advantage of the interpreter and the performance speedup of the JIT compiler can be combined. Such a processor must support the architectural features specified for the JVM. A *Java processor* is an execution model that implements the JVM in silicon to directly execute Java bytecodes. Java processors can be tailored specifically to the Java environment by providing hardware support for such features as stack processing, multithreading, and garbage collection. Thus, a Java processor can potentially deliver much better performance for Java applications than a general-purpose processor. Java processors appear to be particularly well-suited for cost-sensitive embedded computing applications.

Some of the design goals behind the JVM definition were to provide portability, security, and small code size for the executable programs. In addition, it was designed to simplify the task of writing an interpreter or a JIT compiler for a specific target processor and operating system. However, these design goals result in architectural features for the JVM that pose significant challenges in developing an effective implementation of a Java processor [O'Connor and Tremblay 1997]. In particular, there are certain common characteristics of Java programs that are different from traditional procedural programming languages. For instance, Java processors are stack-based and must support the multithreading and unique memory management features of the JVM. These unique characteristics suggest that the architects of a Java processor must take into consideration the dynamic frequency counts of the various instructions types to achieve high performance.

The JVM instructions fall into several categories, namely, local-variable loads and stores, memory loads and stores, integer and floating-point computations, branches, method calls and returns, stack operations, and new object creation. Figure 3 shows the dynamic frequencies of the various JVM instruction categories measured in the Java benchmark programs *LinPack*, *CaffeineMark*, *Dhrystone*, *Symantec*, *JMark2.0*, and *JavaWorld* (benchmark program details are provided in Section 5). These dynamic instruction frequencies were obtained by instrumenting the source code of the Sun 1.1.5 JVM interpreter to count

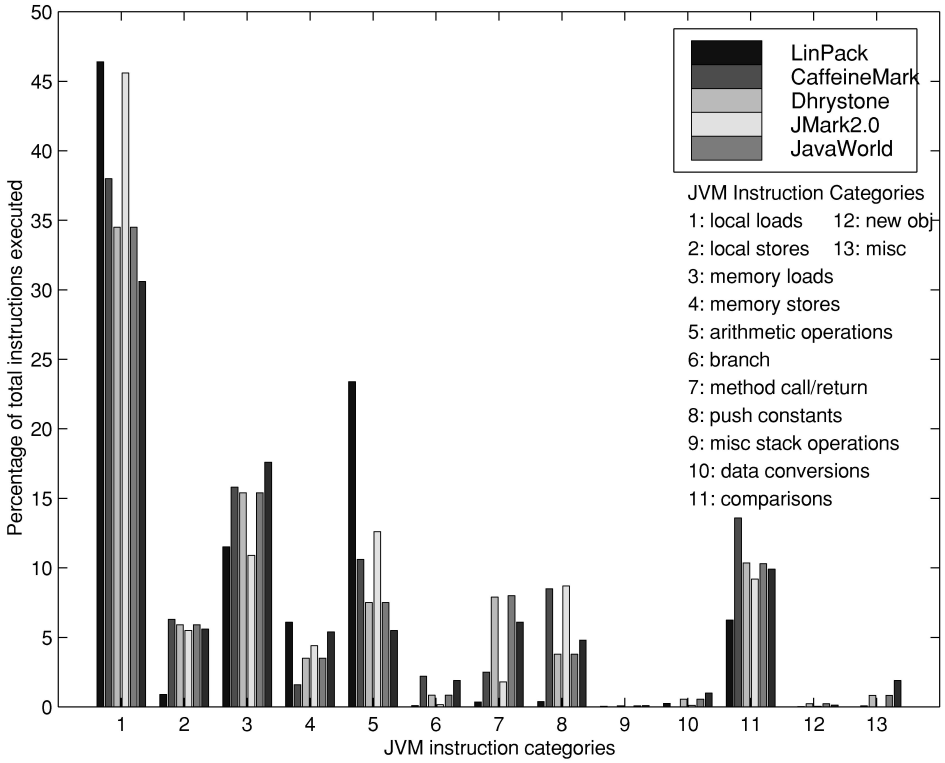

Fig. 3. Dynamic execution frequency of JVM bytecode instructions measured for several common benchmark programs.

the number of times each type of bytecode was executed.

As seen in this figure, the most frequent instructions are local variable loads (about 30–47% of the total instructions executed), which move operands from the local variables area of the stack to the top of the stack. The high frequency of these loads suggests that optimizing local loads could substantially enhance performance. Method calls and returns are also quite common in Java programs. As seen in Figure 3, they constitute about 7% of the total instructions executed. Hence, optimizing the method call and return process is expected to have a relatively large impact on the performance of Java codes. Since method calls occur through the stack, a Java processor also should have an efficient stack implementation.

Another common operation supported by the JVM is the concurrent execution of multiple threads. As with any multithreading execution, threads will often need to enter synchronized critical sections of code to provide mutually exclusive access to shared objects. Thus, a Java processor should provide architectural support for this type of synchronization. A Java processor must also provide support for memory management via the garbage collection process, such as hardware tagging of memory objects, for instance.

PicoJava-I [McGham and O'Connor 1998; O'Connor and Tremblay 1997] is a configurable processor core that supports the JVM specification. It includes a RISC-style pipeline executing the JVM instruction set. However, only the most common instructions that most directly impact the program execution are implemented in hardware. Some moderately complicated but performance critical instructions are implemented through microcode. The remaining instructions are

trapped and emulated in software by the processor core. The hardware design is thus simplified since the complex, but infrequently executed, instructions do not need to be directly implemented in hardware.

In the Java bytecodes, it is common for an instruction that copies data from a local variable to the top of the stack to precede an instruction that uses it. The picoJava core *folds* these two instructions into one by accessing the local variable directly and using it in the operation. This *folding* operation accelerates Java bytecode execution by taking advantage of the single-cycle random access to the stack cache. Hardware support for synchronization is provided to the operating system by using the low-order 2-bits in shared objects as flags to control access to the object.

The *picoJava-2* core [Turley 1997], which is the successor to the picoJava-I processor, augments the bytecode instruction set with a number of extended instructions to manipulate the caches, control registers, and absolute memory addresses. These extended instructions are intended to be useful for non-Java application programs that are run on this Java execution core. All programs, however, still need to be compiled to Java bytecodes first, since these bytecodes are the processor's native instruction set. The pipeline is extended to 6 stages compared to the 4 stages in the picoJava-I pipeline. Finally, the *folding* operation is extended to include two local variable accesses, instead of just one.

Another Java processor is the *Sun microJava 701* microprocessor [Sun Microsystems. MicroJava-701 Processor]. It is based on the picoJava-2 core and is supported by a complete set of software and hardware development tools. The *Patriot PSC1000* microprocessor [Patriot Scientific Corporation] is a general-purpose 32-bit, stack-oriented architecture. Since its instruction set is very similar to the JVM bytecodes, the PSC1000 can efficiently execute Java programs.

Another proposed Java processor [Vijaykrishnan et al. 1998] provides architectural support for direct object manipulation, stack processing, and method invocations to enhance the execution of Java bytecodes. This architecture uses a *virtual address object cache* for efficient manipulation and relocation of objects. Three cache-based schemes—the *hybrid cache*, the *hybrid polymorphic cache*, and the *two-level hybrid cache*—have been proposed to efficiently implement virtual method invocations. The processor uses *extended folding* operations similar to those in the picoJava-2 core. Also, simple, frequently executed instructions are directly implemented in the hardware, while more complex but infrequent instructions are executed via a trap handler.

The *Java ILP* processor [Ebcioglu et al. 1997] executes Java applications on an ILP machine with a Java JIT compiler hidden within the chip architecture. The first time a fragment of Java code is executed, the JIT compiler transparently converts the Java bytecodes into optimized RISC primitives for a Very Long Instruction Word (VLIW) parallel architecture. The VLIW code is saved in a portion of the main memory not visible to the Java architecture. Each time a fragment of Java bytecode is accessed for execution, the processor's memory is checked to see if the corresponding ILP code is already available. If it is, then the execution jumps to the location in memory where the ILP code is stored. Otherwise, the compiler is invoked to compile the new Java bytecode fragment into the code for the target processor, which is then executed.

While Java processors can deliver significant performance speedups for Java applications, they cannot be used efficiently for applications written in any other language. If we want to have better performance for Java applications, and to execute applications written in other languages as well, Java processors will be of limited use. With so many applications written in other languages already available, it may be desirable to have general-purpose processors with enhanced architectural features to support faster execution of Java applications.

4. HIGH-PERFORMANCE JAVA EXECUTION TECHNIQUES

Direct compilers or bytecode-to-source translators can improve Java performance by generating optimized native codes or intermediate language codes. However, this high performance may come at the expense of a loss of portability and flexibility. JIT compilers, on the other hand, support both portability and flexibility, but they cannot achieve performance comparable to directly compiled code as only limited-scope optimizations can be performed. A number of execution techniques have been developed that attempt to improve Java performance by optimizing the Java source code or the bytecodes so that portability is not lost. Some techniques apply dynamic optimizations to the JIT compilation approach to improve the quality of the compiled native machine code within the limited time available to a JIT compiler. Some other techniques improve various features of the JVM, such as thread synchronization, remote method invocation (RMI), and garbage collection, that aid in improving the overall performance of Java programs. In this section, we describe these high-performance Java execution techniques.

4.1. Bytecode Optimization

One technique for improving the execution time performance of Java programs is to optimize the bytecodes. The Briki compiler applies a number of optimizations to bytecodes to improve the execution time performance [Cierniak and Li 1997a,b]. The front-end of the offline mode of the Briki compiler [Cierniak and Li 1997a] reconstructs the high-level program structure of the original Java program from the bytecodes. This high-level view of the input bytecodes is stored in an intermediate representation called JavaIR. A number of optimizations can then be applied to this JavaIR. Finally, the optimized JavaIR is transformed back into Java source code and executed by a Java compiler.

Converting Java bytecodes to JavaIR is very expensive, however, requiring an order of magnitude more time than the time required to perform the optimizations themselves. In an on-line version of Briki [Cierniak and Li 1997b], the same optimizations are performed while recovering only as much of the structure information as needed and using faster analysis techniques than those used in traditional compilers. This on-line version of Briki is integrated with the Kaffe JIT compiler [Wilkinson, Kaffe v0.10.0], using Kaffe to generate an IR. The compiler analyzes and transforms the Kaffe IR into an optimized IR which the Kaffe JIT backend then translates into native code.

One optimization in Briki attempts to improve memory locality by data remapping. Briki groups together similar fields within objects onto consecutive memory locations, increasing the probability that fields will reside on the same cache line. However, the specific criterion used by the Briki compiler to determine similar fields has not been described [Cierniak and Li 1997a,b]. Briki also attempts to remap array dimensions when dimension remapping will improve memory access time. For example, in a two-dimensional array A, location A[i][j] might be remapped to A[j][i] if such a remapping is advantageous. However, this remapping poses a number of problems. First, Java arrays are not standard rectangular arrays but rather are implemented as an array of arrays. Therefore, array elements in a given dimension are not guaranteed to be of the same size. A[i] might reference an array of size 2 while A[j] references an array of size 3, for instance. Since array remapping is valid only for rectangular arrays, Briki constrains remapping to cases where all dimensions of the array are accessed. Second, subscript expressions in arrays can have potential side-effects. For instance, the calculation of a subscript may, as a side-effect, assign a value to a variable. Remapping the array dimensions may cause problems since the renaming changes the evaluation order of these expressions. To remedy this problem, Briki evaluates subscript expressions in the correct order before the arrays are actually referenced.

The *DashO Pro* [DashO Pro] bytecode optimizer applies several optimization techniques to improve runtime performance and reduce the size of the Java executable. These techniques include shortening the names of all the classes, methods, and fields, removing all unused methods, fields, and constant-pool entries (both constants and instance variables), extracting classes from accessible third-party packages or libraries, and so on. DashO produces one file that contains only the class files needed for the Java program resulting in smaller and faster Java bytecode files.

Krintz et al. [1999] proposed a Java class file splitting and prefetching technique to reduce the transfer delay of bytecodes sent over the Internet. Class file splitting partitions a Java class file into separate *hot* and *cold* class files to eliminate transferring code in the cold class file that is very rarely (or never) used. Class file prefetching inserts special prefetch instructions into the Java class file to overlap bytecode transfer with execution. These optimizations use compile-time analysis and profiling to select code to split and to insert prefetch instructions. Experimental results showed that class file splitting was able to reduce startup time for Java programs by 10% while prefetching combined with splitting reduced the overall bytecode transfer delay by 25% on average.

The Java application extractor Jax [Tip et al. 1999] applies several techniques, such as the elimination of redundant methods/fields and class hierarchy specialization [Tip and Sweeney 1997], to reduce the size of a Java class file archive (e.g., zip or jar files). Jax performs a whole-program analysis on a Java class file to determine the classes, methods, and fields that must be retained to preserve program behavior. It then removes the unnecessary components of the class files to reduce the archive size. Additionally, Jax applies a class hierarchy transformation that reduces the archive size by eliminating classes entirely or by merging adjacent classes in the hierarchy. It also replaces original class, method, and field names with shorter names. Once methods are removed from a class file, some entries in the constant pool may appear to be redundant. Thus, when the archive is rewritten after all the transformations are applied, a new constant pool is created from scratch to contain only the classes, methods, fields, and constants that are referenced in the transformed class file. Using Jax on a number of Java archives resulted in a 13.4–90.2% reduction in size [Tip et al. 1999].

4.2. Parallel and Distributed Execution Techniques

A number of techniques parallelize Java source code or bytecodes to improve the execution-time performance of the application. The parallelization typically is achieved through Java language-level support for multithreading. Thus, these techniques maintain the portability of the transformed parallel Java programs. Most of these techniques exploit implicit parallelism in Java programs for parallel execution on shared-memory multiprocessor systems using Java multithreading and synchronization primitives. Some approaches extend the Java language itself to support parallel and distributed Java programs. The performance improvement obtained when using these techniques depends on the amount of parallelism that can be exploited in the application program.

The *High Performance Java* project [Bik and Gannon 1997; Bik and Gannon 1998] exploits implicit parallelism in loops and multiway recursive methods to generate parallel code using the standard Java multithreading mechanism. The *JAVAR* [Bik and Gannon 1997] tool, which is a source-to-source restructuring compiler, relies on explicit annotations in a sequential Java program to transform a sequential Java source code into a corresponding parallel code. The transformed program can be compiled into bytecodes using any standard Java compiler. The *JAVAB* [Bik and Gannon 1998] tool, on the other hand, works directly on Java bytecodes to automatically detect and exploit

implicit loop parallelism. Since the parallelism is expressed in Java itself using Java's thread libraries and synchronization primitives, the parallelized bytecodes can be executed on any platform with a JVM implementation that supports native threads.

The *Java Speculative Multithreading* (JavaSpMT) parallelization technique [Kazi and Lilja 2000] uses a speculative thread pipelining execution model to exploit implicit loop-level parallelism on shared-memory multiprocessors for general-purpose Java application programs. Its support of control speculation combined with its run-time data-dependence checking allows JavaSpMT to parallelize a wide variety of loop constructs, including *do–while* loops. JavaSpMT is implemented using the standard Java multithreading mechanism. The parallelism is expressed by a Java source-to-source transformation.

The *Do!* project [Launay and Pazat 1997] provides a parallel framework embedded in Java to ease parallel and distributed programming in Java. The parallel framework supports both data parallelism and control (or task) parallelism. The framework provides a model for parallel programming and a library of generic classes. Relevant library classes can be extended for a particular application or new framework classes can be defined for better tuning of the application.

Tiny Data-Parallel Java [Ichisugi and Roudier 1997] is a Java language extension for data-parallel programming. The language defines data-parallel classes whose methods are executed on a large number of virtual processors. An extensible Java preprocessor, EPP, is used to translate the data-parallel code into standard Java code using the Java thread libraries and synchronization primitives. The preprocessor can produce Java code for multiprocessor and distributed systems as well. However, the Tiny Data-Parallel Java language does not yet have sufficient language features to support high-performance parallel programs. *DPJ* [Ivannikov et al. 1997] defines a parallel framework through a Java class library for the development of data-parallel programs.

4.3. Dynamic Compilation

Since the compilation time in JIT compilation adds directly to the application's total execution time, the quality of the code optimization is severely constrained by compilation speed. Dynamic compilation addresses this problem of JIT compilation by optimizing only the portions of the code that are most frequently executed, i.e., program *hotspots*. Most programs spend the majority of the time executing only a small fraction of their code. Thus, optimizing only the hotspot methods should yield a large performance gain while keeping the compilation speed relatively fast.

Sun's *Hotspot* JVM [The Java Hotspot Performance Engine Architecture] uses dynamic compilation to generate optimized native machine code during run-time. The Hotspot engine contains both a run-time compiler and an interpreter. The first time a method is executed, it is interpreted using a profiling interpreter that gathers run-time information about the method. This information is used to detect hotspots in the program and to gather information about program behavior that can be used to optimize generated native code in later stages of program execution. After the hotspot methods are identified, they are dynamically compiled to generate optimized native machine code. Infrequently executed code continues to be interpreted, decreasing the amount of time and memory spent on native code generation. Because a program is likely to spend the majority of its execution time in the hotspot regions detected by the interpreter, the compiler can spend more time optimizing the generated code for these sections of the program than a JIT-compiler while still producing an overall improvement in execution time. During code generation, the dynamic compiler performs conventional compiler optimizations, Java specific optimizations, and inlining of static and dynamic methods. The inlining optimizations are designed to be

reversible due to the problems associated with dynamic class loading.

IBM's *Jalapeno* JVM [Burke et al. 1999] includes an adaptive dynamic optimizing compiler that generates optimized machine code as the program is executed. The Jalapeno JVM does not use an interpreter. Instead, the first execution of a method is handled by quickly compiling a method into an unoptimized executable code. The dynamic optimizing compiler later generates optimized executable code from the bytecodes of the hotspot (i.e., frequently executed) methods as determined by runtime profile information.

The Jalapeno optimizing compiler first compiles Java bytecodes into a virtual register-based high-level intermediate representation (HIR). It then generates a control flow graph for the method. After applying compiler optimizations on the HIR, the intermediate executable is converted to a low-level representation (LIR) that includes references to specific implementation details, such as parameter passing mechanisms and object layouts in memory. The LIR is then optimized further and converted to a native code representation using a Bottom-Up Rewrite System [Proebsting 1992] approach. Finally, this machine specific representation (MIR) is optimized and assembled into machine executable code. The optimizing compiler performs inlining both during bytecode-to-IR translation and during HIR optimization. Inlining of virtual functions is handled by predicting the type of the virtual function and performing a run-time check to determine the validity of the prediction. If the prediction is incorrect, the program performs a normal virtual function invocation. Otherwise, the program proceeds with the inlined function.

The program adaptation in Jalapeno starts with the instrumentation and recompilation of executable code. The executable code is instrumented to gather context sensitive profile information to aid optimization. The profile information is used to detect program hotspots. When a certain performance threshold is reached, the dynamic optimizing compiler is invoked to recompile the hotspot methods using context specific optimizations at all levels of representations (HIR, LIR, and MIR). The unoptimized code is then replaced by optimized code based on the collected profile information. Program adaptation continues in this cycle, with executable code being improved on every optimization iteration.

4.4. Improved JVM Features

The underlying implementation of various JVM features, such as thread synchronization, RMI support, and garbage collection, often has a significant effect on the execution time performance of Java application programs. While these features make Java more powerful as a programming language, they tend to add delays to a program's execution time. Hence, it is important to efficiently implement these features to improve Java performance. Furthermore, Java, as it is now, is not suitable for high-performance numerical computing. To support numerical applications, Java should include features that allow efficient execution of floating point and complex numbers.

4.4.1. Thread Synchronization

Thread synchronization is a potential performance problem in many Java programs that use multithreading. Since Java libraries are implemented in a thread-safe manner, the performance of even single-threaded applications may be degraded due to synchronization. In Java, synchronization is provided through monitors, which are language-level constructs used to guarantee mutually-exclusive access to shared data-structures [Silberschatz and Galvin 1997]. Unfortunately, monitors are not efficiently implemented in the current Sun JDK. Since Java allows any object to be synchronizable (with or without any synchronized methods), using a lock structure for each object can be very costly in terms of memory. To minimize memory requirements, the Sun JDK keeps monitors outside of the objects. This requires the run-time system to first query each monitor in a monitor cache before it

is used, which is quite inefficient. Further, the monitor cache itself needs to be locked during these queries to avoid race conditions. Thus, this monitor implementation approach is not scalable.

CACAO, in contrast, is a system that implements monitors using a hash table indexed by the address of the associated object [Krall and Proebst 1998]. Because only a small number of objects that require mutually-exclusive access (called a *mutex* object) are locked at any given time, this approach generally leads to decreased memory usage. Each of these objects maintains a count of how many lock operations have been performed on it without any corresponding unlock operations. When the count goes to zero, the lock is not owned by any thread. A mutex object, however, is not destroyed when its count reaches zero under the assumption that the mutex object will be reused again soon. The mutex object is destroyed only when a new mutex object collides with it in the hash table. This memory allocation approach has the advantage of requiring less code to implement lock and unlock operations. Since the hash table entries will be free only at the beginning of the program and will never be freed again, the code to lock the mutex need not check if the entry is free. The unlocking code also does not need to check whether the hash entry should be freed.

The *thin locks* approach in IBM's JDK 1.1.2 for AIX improves thread synchronization by optimizing lock operations for the most common cases [Bacon et al. 1998]. *Thin locks*, which require only a partial word per object, are used for objects that are not subject to any contention, do not have wait/notify operations performed on them, and are not locked to an excessive nesting depth. Objects that do not meet the criterion use *fat locks*, which are multiword locks similar to those used in the standard Java implementation. To minimize the memory required for each object's lock, 24 bits in the object's header are used for the lock structure. The other values stored in the header are compacted using various encoding schemes to make the 24 bits available for this lock structure. This 24-bit field stores either a thin lock directly, or a pointer to a fat lock. The dedicated lock for each object allows the lock/unlock operations on the thin locks to be performed using only a few machine instructions. This approach also eliminates the need to synchronize the monitor cache.

4.4.2. Remote Method Invocation

The Java Remote Method Invocation (RMI) [Remote Method Invocation Specification] mechanism enables distributed programming by allowing methods of remote Java objects to be invoked from other JVMs, possibly on different physical hosts. A Java program can invoke methods on a remote object once it obtains a reference to the remote object. This remote object reference is obtained either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. RMI uses object serialization to marshal and unmarshal parameters.

The current Java RMI is designed to support client-server applications that communicate over TCP-based networks [Postel 1981]. Some of the RMI design goals, however, result in severe performance limitations for high-performance applications on closely connected environments, such as clusters of workstations and distributed memory processors. The Applications and Concurrency Working Group (ACG) of the Java Grande Forum (JGF) [Java Grande Forum Report] assessed the suitability of RMI-based Java for parallel and distributed computing based on Java RMI. JGF proposed a set of recommendations for changes in the Java language, Java libraries, and JVM implementation to make Java more suitable for high-end computing. ACG emphasized improvements in two key areas, object serialization and RMI implementation, that will potentially improve the performance of parallel and distributed programs based on Java RMI.

Experimental results [Java Grande Forum Report] suggest that up to 30% of the execution time of a Java RMI is spent

in object serialization. Accordingly, ACG recommended a number of improvements, including a slim encoding technique for type information, more efficient serialization of float and double data types, and enhancing the reflection mechanism to allow fewer calls to be made to obtain information about a class. In the current RMI implementation, a new socket connection is created for every remote method invocation. In this implementation, establishing a network connection can take up to 30% of the total execution of the remote method [Java Grande Forum Report]. The key recommendations of JGF to reduce the connection overhead include improved socket connection management, improved resource management, and improved support for custom transport.

4.4.3. Java for High-Performance Numeric Computation

The current design and implementation of Java does not support high-performance numerical applications. To make the performance of Java numerical programs comparable to the performance obtained through programming languages such as C or Fortran, the numeric features of the Java programming language must be improved. The Numerics Working Group of the Java Grande Forum assessed the suitability of Java for numerical programming and proposed a number of improvements in Java's language features, including floating-point and complex arithmetic, multidimensional arrays, lightweight classes, and operator overloading [Java Grande Forum Report].

To directly address these limitations in Java, IBM has developed a special library that improves the performance of numerically-intensive Java applications [Moreira et al. 2000]. This library takes the form of the Java Array package and is implemented in the IBM HPCJ. The Array package supports such FORTRAN 90 like features as complex numbers, multidimensional arrays, and linear algebra library. A number of Java numeric applications that used this library were shown to achieve between 55% and 90% of the performance of corresponding highly optimized FORTRAN codes.

4.4.4. Garbage Collection

Most JVM implementations use conservative garbage collectors that are very easy to implement, but demonstrate rather poor performance. Conservative garbage collectors cannot always determine where all object references are located. As a result, they must be careful in marking objects as candidates for garbage collection to ensure that no objects that are potentially in use are freed prematurely. This inaccuracy sometimes leads to memory fragmentation due to the inability to relocate objects.

To reduce the negative performance impacts of garbage collection, Sun's Hotspot JVM [The Java Hotspot Performance Engine Architecture] implements a fully accurate garbage collection (GC) mechanism. This implementation allows all inaccessible memory objects to be reclaimed while the remaining objects can be relocated to eliminate memory fragmentation. Hotspot uses three different GC algorithms to efficiently handle garbage collection. A *generational* garbage collector is used in most cases to increase the speed and efficiency of garbage collection. Generational garbage collectors cannot, however, handle long-lived objects. Consequently, Hotspot needs to use an old-object garbage collector, such as a *mark-compact* garbage collector to collect objects that accumulate in the "Old Object" area of the generational garbage collector. The old-object garbage collector is invoked when very little free memory is available or through programmatic requests.

In applications where a large amount of data is manipulated, longer GC pauses are encountered when a *mark-compact* collector is used. These large latencies may not be acceptable for latency-sensitive or data-intensive Java applications, such as server applications and animations. To solve this problem, Hotspot provides an alternative *incremental* garbage collector to collect objects in the "Old Object" area. Incremental garbage collectors can

Table 1. The Benchmark Programs in SPEC JVM98 Benchmark Suite

check	A simple program to test various features of the JVM
compress	A modified Lempel-Ziv (LZW) compression algorithm
jess	A Java Expert Shell System (JESS) based on NASA's CLIPS expert shell system
db	A database application that performs multiple database functions on a memory resident database
javac	The Java compiler from Sun's JDK 1.0.2
mpegaudio	An application that decompresses audio files conforming to the ISO MPEG Layer-3 audio specification
mtrt	A ray tracer program that works on a scene depicting a dinosaur, where two threads each render the scene in the input file
jack	A Java parser generator that is based on the Purdue Compiler Construction Tool Set (PCCTS)

potentially eliminate all user-perceived GC pauses by interleaving the garbage collection with program execution.

5. BENCHMARKS

Appropriate benchmarks must be identified to effectively evaluate and compare the performance of the various Java execution techniques. A benchmark can be thought of as a test used to measure the performance of a specific task [Lilja 2000]. As existing execution techniques are improved and new techniques are developed, the benchmarks serve as a vehicle to provide consistency among evaluations for comparison purposes. The problem that exists in the Java research community is that no benchmarks have been agreed upon as the standard set to be used for Java performance analysis. Instead, researchers have used a variety of different benchmarks making it very difficult to compare results and determine which techniques yield the best performance. In this section, we identify some of the popular benchmarks that have been used to compare various Java execution techniques.

Application benchmarks are used to evaluate the overall system performance. *Microbenchmarks*, on the other hand, are used to evaluate the performance of individual system or language features, such as storing an integer in a local variable, incrementing a byte, or creating an object [Java Microbenchmarks, UCSD Benchmarks for Java].

The Open Systems Group (OSG) of the Standard Performance Evaluation Corporation (SPEC) [SPEC JVM98 Benchmarks] developed a Java benchmark suite, *SPEC JVM98*, to measure the performance of JVMs. The benchmarks in this suite are expected to serve as a standard for performance evaluation of different JVM implementations to provide a means for comparing the different techniques using a common basis. SPEC JVM98 is a collection of eight general-purpose Java application programs, as listed in Table 1.

SYSmark J [SYSmark] is another benchmark suite that consists of four Java applications: *JPhoto Works*, which is an image editor using filters, the *JNotePad* text editor, *JSpreadSheet*, which is a spreadsheet that supports built-in functions in addition to user-defined formulas; and the *MPEG* video player.

A series of microbenchmark tests, known as *CaffeineMark* [CaffeineMark], has been developed to measure specific aspects of Java performance, including loop performance (*loop test*), the execution of decision-making instructions (*logic test*), and the execution of recursive function calls (*method test*). Each of the tests is weighted to determine the overall CaffeineMark score. However, since the CaffeineMark benchmark does not emulate the operations of real-world programs [Armstrong 1998], there is a question as to whether this benchmark makes reliable measurements of performance. The *Embedded CaffeineMark benchmark*

[CaffeineMark] is similar to the CaffeineMark benchmark except that all graphical tests are excluded. This benchmark is intended as a test for embedded systems.

The *JMark* benchmark [PC Magazine Test Center, Welcome to JMark 2.0], from PC Magazine, consists of eleven synthetic microbenchmarks used to evaluate Java functionality. Among the tests are the graphics mix test, stack test, bubble sort test, memory allocation and garbage collection test, and the graphics thread test. JMark is one of the few benchmarks that attempt to evaluate multithreading performance. *VolanoMark* [Java Benchmarks] is a new server-side Java benchmarking tool developed to assess JVM performance under highly multithreaded and networked conditions. It generates ten groups of 20 connections each with a server. Ten messages are broadcast by each client connection to its respective group. VolanoMark then returns the average number of messages transferred per second as the final score.

The Java Grande Forum has developed a benchmark suite, called the *Java Grande Forum Benchmark Suite* [Java Grande Forum Benchmark Suite], that consists of benchmarks in three categories. The first category, the *low-level operations*, measures the performance of such JVM operations as arithmetic and math library operations, garbage collection, method calls and casting. The benchmarks in the *kernel* category are small program kernels, including *Fourier coefficient analysis*, *LU factorization*, *alpha–beta pruned search*, *heapsort*, and *IDEA encryption*. The *Large Scale Applications* category is intended to represent real application programs. Currently, the only application program in this category is *Euler*, which solves the time-dependent Euler equations for flow in a channel.

The *Linpack* benchmark is a popular Fortran benchmark that has been translated into C and is now available in Java [Linpack Benchmark]. This numerically intensive benchmark measures the floating-point performance of a Java system by solving a dense system of linear equations, $A\,x = b$. *JavaLex* [JLex] (a lexical analyzer generator), *JavaCup* [CUP Parser Generator for Java] (a Java parser generator), *JHLZip* [JHLZip] (combines multiple files into one archive file with no compression), and *JHLUnzip* [JHLUnzip] (extracts multiple files from JHLZip archives) are some other Java application benchmarks. Other available benchmarks include *JByte*, which is a Java version of the *BYTEmark* benchmark available from BYTE magazine [TurboJ Benchmark's Results]; the *Symantec* benchmark, which incorporates tests such as *sort* and *sieve* [Symantec, TurboJ Benchmark's Results]; the *Dhrystone* CPU benchmark [Dhrystone Benchmark, Dhrystone Benchmark in Java]; *Jell*, a parser generator [Jell]; *Jax*, a scanner generator [Jax]; *Jas*, a bytecode assembler [Jas]; and *EspressoGrinder*, a Java source program-to-bytecode translator [EspressoGrinder]. Additional benchmark references are available on the Java Grande Forum Benchmark website [Java Grande Forum Benchmark].

To compare the performance of the various Java execution techniques, a standard set of benchmarks must be used. The benchmarks must be representative of real-world applications to provide any useful performance evaluations. Many of the Java benchmarks discussed earlier, such as CaffeineMark, Jmark, and Symantec, consist of small programs designed to test specific features of a JVM implementation. Since they do not test the performance of the JVM as a whole, the performance results obtained through these benchmarks will be of little significance when the JVM is used to execute real applications. Some benchmarks, such as JHLZip and JHLUnzip, are I/O bound applications that are not suitable for measuring the performance obtained through compiler optimizations. The SPEC JVM98 suite includes several applications, such as a compiler, a database, and an mpegaudio application, that represent real applications very well. Benchmarks such as these will, therefore, help in understanding the potential performance of various

Table 2. Reported Relative Performance of Different Java Execution Techniques and Compiled C

Interpreters Sun JDK	Interpreters (w.r.t.)	JIT compilers (w.r.t.)	C performance 10–50× slower
JIT Compilers			
Microsoft JIT Visual J++ v1.0	5.6× faster (SUN JDK1.0.2)		4× slower
Symantec Cafe version 1.0	5× faster (SUN JDK1.0.2)		
IBM JIT version 3.0	6.6× faster (IBM JDK1.16)		
Kaffe version 0.9.2	2.4× faster (SUN JDK1.1.1)		
CACAO	5.4× faster (SUN JDK1.1)		1.66× slower
AJIT	3.5× faster (SUN JDK1.1.1)	1.46× faster (Kaffe v0.9.2)	
Direct Compilers			
NET	17.3× faster (SUN JDK1.0.2)	3× faster (MS JIT v1.0)	1.85× slower
Caffeine	20× faster (SUN JDK1.0.2)	4.7× faster (Symantec Cafe)	1.47× slower
HPCJ	10× faster (IBM AIX1.0.2)	6× faster (IBM JIT v1.0)	
Bytecode Translators			
Toba	5.4× faster (SUN JDK1.0.2)	1.6× faster (Guava 1.0b1)	
Harissa	5× faster (SUN JDK1.0.2)	3.2× faster (Guava 1.0b4)	
Java Processors			
pica Java-I	12× faster (SUN JDK1.0.2)	5× faster (Symantec Cafe 1.5)	

This table shows that the Caffeine compiler produces code that is approximately 20 times faster than the Java code interpreted with Sun's JDK1.0.2 JVM, approximately 4.7 times faster than the Symantec Cafe JIT compiler, and approximately 1.47 times slower than an equivalent compiled C program.

Java execution techniques when used in a real programming environment.

6. PERFORMANCE COMPARISON

Several studies have attempted to evaluate the different execution techniques for Java programs using various combinations of the benchmarks mentioned in Section 5. The performance evaluation of the different execution techniques are incomplete, however, due to the lack of a standardized set of Java benchmarks. Also, many of the techniques that have been evaluated are not complete implementations of the JVM and the benchmarks have been run on a wide variety of underlying hardware configurations. While incomplete and difficult to compare directly, these performance results do provide a limited means of comparing the alternative techniques to determine which ones may potentially provide an improvement in performance and to what degree.

Table 2 summarizes the relative performance of the various techniques based on reported results. However, the numbers alone do not prove the superiority of one technique over another. When comparing alternative techniques for Java execution, it is important to consider the implementation details of each technique. To support a complete implementation of the JVM, any runtime approach should include garbage collection, exception handling, and thread support. However, each of these features can produce possibly substantial execution overhead. Hence, a technique that includes garbage collection

Table 3. Support for Exception Handling, Multithreading, and Garbage Collection as Reported for the Various Stand-alone Java Execution Techniques Compared in Table 2

	Exception handling	Multithreading	Garbage collection
NET	Yes	Yes	Yes
Caffeine	Yes	No	No
HPCJ	Yes	Yes	Yes
Toba	Yes	Yes	Yes
Harissa	Yes	No	Yes
picoJava-I	Yes	Yes	Yes

will require more time to execute a Java code than a similar technique that does not support garbage collection. The reader is cautioned that some of the reported values apparently do not include these overhead effects. Table 3 lists whether each execution technique includes support for exception handling, multithreading, and garbage collection. Furthermore, the performance evaluations used a variety of different benchmarks making it difficult to directly compare the different execution techniques.

At present, the performance of Java interpreters is typically about 10–50 times slower than compiled C performance [Java Compiler Technology]. The Microsoft JIT compiler included in Visual J++ v2.0 has been reported to achieve approximately 23% of "typical" compiled C performance. That is, this JIT compiler produces code that executes approximately 4 times slower than compiled C code. When compared to the Sun JDK1.0.2 interpreter, however, an average speedup of 5.6 was obtained [Hsieh et al. 1997]. This performance evaluation was performed as part of evaluating the NET direct compiler using a variety of benchmark programs, in particular, the Unix utilities *wc, cmp, des,* and *grep*; Java versions of the SPEC benchmarks *026.compress, 099.go, 130.li*, and *129.compress*; the C/C++ codes *Sieve, Linpack, Pi, othello,* and *cup*; and the *javac* Java bytecode compiler.

The Symantec Cafe JIT compiler has been shown to be about 5 times faster, on average, than the Sun JDK1.0.2 interpreter based on the Java version of the C programs *cmp, grep, wc, Pi, Sieve*, and *compress* [Hsieh et al. 1996]. The IBM JIT compiler v3.0, which applies an extensive set of optimizations, performs about 6.6 times faster than IBM's enhanced port of Sun JDK1.1.6 [Burke et al. 1999]. The performance analysis was performed using four SPEC JVM98 benchmark programs—*compress, db, javac*, and *jack*.

Kaffe (version 0.9.2) typically performs about 2.4 times faster than the Sun JDK1.1.1 interpreter for the benchmarks *Neighbor* (performs a nearest-neighbor averaging), *EM3D* (performs electromagnetic simulation on a graph), *Huffman* (performs a string compression/decompression algorithm), and *Bitonic Sort* [Azevedo et al. 1999].

CACAO improves Java execution time by a factor of 5.4, on average, over the Sun Java interpreter [Krall and Grafl 1997]. These performance results were obtained using the benchmark programs *JavaLex, javac, espresso, Toba*, and *java_cup*. When compared to compiled C programs optimized at the highest level, though, CACAO (with array bounds check and precise floating point exception disabled) performs about 1.66 times slower. This comparison with compiled C programs was performed using the programs *Sieve, addition*, and *Linpack*. Since these are very small programs, the performance difference compared to compiled C code is likely to be higher when applied to real applications.

The AJIT compiler performs about 3.4 times better than the Sun JDK1.1.1 interpreter and about 1.46 times faster than the Kaffe JIT compiler for the benchmarks *Neigbor, EM3D, Huffman*, and *Bitonic Sort* [Azevedo et al. 1999].

Performance evaluation results of a number of direct compilers show their higher performance compared to the interpreters and JIT compilers. The NET compiler, for instance, achieves a speedup of up to 45.6 when compared to the Sun JDK1.0.2 interpreter, with an average speedup of about 17.3 [Hsieh et al. 1997]. Compared to the Microsoft JIT, NET is about 3 times faster, on average, and currently achieves 54% of the performance of compiled C code. The NET compiler uses a mark-and-sweep based garbage collector. The overhead of the garbage collection is somewhat reduced by invoking the garbage collector only when the memory usage reaches some predefined limit. The benchmarks used in evaluating NET include the Unix utilities *wc, cmp, des,* and *grep*; Java versions of the SPEC benchmarks *026.compress, 099.go, 132.ijpeg*, and *129.compress*; the C/C++ codes *Sieve, Linpack, Pi, othello,* and *cup*; and the Java codes *JBYTEmark* and *javac*.

The Caffeine compiler yields, on average, 68 percent of the speed of compiled C code [Hseih et al. 1996]. It runs 4.7 times faster than the Symantec Cafe JIT compiler and 20 times faster than the Sun JDK1.0.2 interpreter. The reported performance of Caffeine is better than the NET compiler since it does not include garbage collection and thread support. The benchmarks used are *cmp, grep, wc, Pi, Sieve,* and *compress*.

The performance results obtained for the HPCJ show an average speedup of 10 over the IBM AIX JVM1.0.2 interpreter [Seshadri 1997]. HPCJ is about 6 times faster than the IBM AIX JIT compiler (version 1.0). The HPCJ results include the effects of a conservative garbage collection mechanism, exception handling, and thread support. Its performance was evaluated using a number of language processing benchmarks, namely, *javac*, *JacorB* (a CORBA/IDL to Java translator), *Toba, JavaLex, JavaParser, JavaCup,* and *Jobe*.

Codes generated by the bytecode compiler Toba when using a conservative garbage collector, thread package, and exception handling, are shown to run 5.4 times faster than the Sun JDK1.0.2 interpreter and 1.6 times faster than the Guava JIT compiler [Proebsting et al. 1997]. The results are based on the benchmarks *JavaLex, JavaCup, javac, espresso, Toba, JHLZip,* and *JHLUnzip*.

The Harissa bytecode-to-C translator performs about 5 times faster than the Sun JDK1.0.2 interpreter and 3.2 times faster than the Guava JIT compiler based on the *javac* and *javadoc* benchmark programs [Muller et al. 1997]. This version of Harissa includes a conservative garbage collector but does not provide thread support.

In addition to analyzing the performance of the various interpreters and compilers, the improvement in performance due to the use of Java processors has also been considered. The performance of the picoJava-I core was analyzed by executing two Java benchmarks, *javac* and *Raytracer*, on a simulated picoJava-I core [O'Connor and Tremblay 1997]. The results show that the simulated picoJava-I core executes the Java codes about 12 times faster than the Sun JDK1.0.2 interpreter, and about 5 times faster than the Symantec Cafe v1.5 JIT compiler executing on a Pentium processor, at an equal clock rate. In the simulation of the picoJava-I core, the effect of garbage collection was minimized by allocating a large amount of memory to the applications.

A simulation-based performance analysis of the proposed Java processor [Vijaykrishnan et al. 1998] shows that the virtual address object cache reduces up to 1.95 cycles per object access compared to the serialized handle and object lookup scheme. The extended folding operation feature eliminates redundant loads and stores that constitute about 9% of the dynamic instructions of the benchmarks studied (*javac, javadoc, disasmb, sprdsheet,* and *JavaLex*).

To roughly summarize this array of performance comparisons, assume we have a Java program that takes 100 seconds to execute with an interpreter. A JIT compiler would execute this program

in 10–50 seconds. A bytecode translator would likely produce an execution time of approximately 20 seconds. A direct Java to native code compilation would result in an execution time of around 5 seconds running on the same hardware as the interpreter. A Java processor implemented with an equivalent technology is likely to record an execution time of 5–10 seconds. Finally, an equivalent C program running on the same hardware would complete in 2–10 seconds.

7. CONCLUSION

Java, as a programming language, offers enormous potential by providing platform independence and by combining a wide variety of language features found in different programming paradigms, such as, an object-orientation model, multithreading, automatic garbage collection, and so forth. The same features that make Java so attractive, however, come at the expense of very slow performance. The main reason behind Java's slow performance is the high degree of hardware abstraction it offers. To make Java programs portable across all hardware platforms, Java source code is compiled to generate platform-independent bytecodes. These bytecodes are generated with no knowledge of the native CPU on which the code will be executed, however. Therefore, some translation or interpretation must occur at run-time, which directly adds to the application program's execution time.

Memory management through automatic garbage collection is an important feature of the Java programming language since it releases programmers from the responsibility for deallocating memory when it is no longer needed. However, this automatic garbage collection process also adds to the overall execution time. Additional overhead is added by such features as exception handling, multithreading, and dynamic loading.

In the standard interpreted mode of execution, Java is about 10–50 times slower than an equivalent compiled C program. Unless the performance of Java programs becomes comparable to compiled programming languages such as C or C++, however, Java is unlikely to be widely accepted as a general-purpose programming language. Consequently, researchers have been developing a variety of techniques to improve the performance of Java programs. This paper surveyed these alternative Java execution techniques. Although some of these techniques (e.g., direct Java compilers) provide performance close to compiled C programs, they do so at the possible expense of the portability and flexibility of Java programs. Java bytecode-to-source translators convert bytecodes to an intermediate source code and attempt to improve performance by applying existing optimization techniques for the chosen intermediate language. Some other techniques attempt to maintain portability by applying standard compiler optimizations directly to the Java bytecodes. Only a limited number of optimization techniques can be applied to bytecodes, though, since the entire program structure is unavailable at this level. Hence, bytecode optimizers may not provide performance comparable to direct compilation.

Another technique to maintain portability while providing higher performance is to parallelize loops or recursive procedures. However, the higher performance of such techniques is obtained only in multiprocessor systems and only on application programs that exhibit significant amounts of inherent parallelism. Yet another approach to high performance Java is a Java processor that directly executes the Java bytecodes as its native instruction set. Although these processors will execute Java programs much faster than either interpreters or compiled programs, they are of limited use since applications written in other programming languages cannot be run efficiently on these processors.

The current state-of-the-art research in Java execution techniques is being pursued along several directions. To match the performance of compiled programs, Java bytecodes must be translated to native machine code. However, the generation of this native machine code must be done dynamically during the program's

execution to support Java's portability and flexibility. Researchers are continually improving JIT compilation techniques to generate more highly optimized code within the limited amount of time available for the JIT compilation. Dynamic compilation techniques based on program execution profiles are also being applied. These techniques dynamically generate highly optimized native code as the program is executed. This dynamic compilation allows the program execution to adapt itself to its varying behavior to thereby provide improved performance.

Another trend for obtaining greater levels of performance is to improve the performance of the various individual JVM features, such as garbage collection, thread synchronization, and exception handling, that add overhead directly to the execution of the native code. Yet another trend is to execute Java programs in parallel and distributed environments. Different parallelization models are being developed to extract parallelism from serial Java programs without modifying the underlying JVM implementation. The bottleneck in this case, however, is the performance of the Java RMI and multithreading mechanisms, although research is also being pursued to provide efficient implementations of Java RMI and thread support mechanisms that will improve the performance of parallel and distributed Java programs.

The choice of a particular Java execution technique must be guided by the requirements of the application program as well as the performance offered by the technique. However, as was pointed out earlier, the performance evaluation of these various execution techniques is incomplete due to the lack of a standardized set of Java benchmark programs. Also, many of the techniques that have been evaluated were not complete implementations of the JVM. Some included garbage collection and exception handling, for instance, while others did not. These methodological variations make it extremely difficult to compare one technique to another even with a standard benchmark suite.

While Java has tremendous potential as a programming language, there is a tremendous amount yet to be done to make Java execution-time performance comparable to more traditional approaches.

ACKNOWLEDGMENTS

We thank Amit Verma and Shakti Davis for their help in gathering some of the information used in this paper.

REFERENCES

ARMSTRONG, E. 1998. HotSpot: A new breed of virtual machine. *Java World*, Mar.

ARNOLD, K. AND GOSLING, J. 1996. *The Java Programming Language*. Addison-Wesley, Reading, MA.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA.

AZEVEDO, A., NICOLAU, A., AND HUMMEL, J. 1999. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Conference on Java Grande*, 142–151.

ADL-TABATABAI, A., CIERNIAK, M., LUEH, G. Y., PARIKH, V. M., AND STICHNOTH, J. M. 1998. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 280–290.

BIK, A. AND GANNON, D. 1997. Automatically exploiting implicit parallelism in Java. In *Concurrency: Practice and Experience*, vol. 9, no. 6, 579–619.

BIK, A. AND GANNON, D. 1998. Javab—A prototype bytecode parallelization tool, *ACM Workshop on Java for High-Performance Network Computing*.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 258–268.

BURKE, M. G., CHOI, J.-D, FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M., SREEDHAR, V. C., AND SRINIVASAN, H. 1999. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Conference on Java Grande*, 129–141.

ByteMark, http://www.byte.com.

CIERNIAK, M. AND LI, W. 1997a. Optimizing Java bytecodes. In *Concurrency: Practice and Experience*, vol. 9, no. 6, 427–444.

CIERNIAK, M. AND LI, W. 1997b. Just-in-time optimizations for high-performance Java programs. In *Concurrency: Practice and Experience*, vol. 9, no. 11, 1063–1073.

CHANG, P. P., MAHLKE, S. A., CHEN, W. Y., WATER, N. J., AND HWU, W. W. 1991. IMPACT: An architectural framework for multiple-instruction-

issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 28, 266–275.

CaffeineMark, http://www.pendragon-software.com/pendragon/cm3/info.html.

CUP Parser Generator for Java, http://www.cs.princeton.edu/~appel/modern/java/CUP.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of ECOOP '95*, vol. 952 of Lecture Notes in Computer Science. Springer-Verlag, New York, 77–101.

DEAN, J., DEFOUW, G., GROVE, D., LITVINOV, V., AND CHAMBERS, C. 1996. Vortex: An optimizing compiler for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 93–100.

DashO Pro, http://www.preemptive.com/DashO/.

Dhrystone Benchmark, http://www.netlib.org/benchmark/dhry-c.

Dhrystone Benchmark in Java, http://www.c-creators.co.jp/okayan/DhrystoneApplet/.

EBCIOGLU, K., ALTMAN, E., AND HOKENEK, E. 1997. A Java ILP machine based on fast dynamic compilation. In *International Workshop on Security and Efficiency Aspects of Java*, Eilat, Israel, Jan. 9–10.

EspressoGrinder, http://wwwipd.ira.uka.de/~espresso/.

FAJITA FAJITA Compiler Project, http://www.ri.silicomp.fr/adv-dvt/java/fajita/index-b.htm.

GOSLING, J., JOY, B., and STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley, Reading, MA.

HSIEH, C. A., GYLLENHAAL, J. C., AND HWU, W. W. 1996. Java bytecode to native code translation: The caffeine prototype and preliminary results. In *International Symposium on Microarchitecture MICRO* 29, 90–97.

HSIEH, C. A., CONTE, M. T., GYLLENHAAL, J. C., AND HWU, W. W. 1997. Optimizing NET compilers for improved Java performance. *Computer*, June, 67–75.

ICHISUGI, Y. AND ROUDIER, Y. 1997. Integrating data-parallel and reactive constructs into Java. In *Proceedings of France–Japan Workshop on Object-Based Parallel and Distributed Computation (OBPDC '97)*, France.

IVANNIKOV, V., GAISSARAYAN, S., DOMRACHEV, M., ETCH, V., AND SHTALTOVNAYA, N. 1997. *DPJ: Java Class Library for Development of Data-Parallel Programs*. Institute for Systems Programming, Russian Academy of Sciences, http://www.ispras.ru/~dpj/.

ISHIZAKI, K., KAWAHITO, M., YASUE, T., TAKEUCHI, M., OLASAWARA, T., SULANUMA, T., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 1999. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 Conference on Java Grande*, June, 119–128.

Jas: Bytecode Assembler, http://www.meurrens.org/ip-Links/Java/codeEngineering/blackDown/jas.html.

Java Benchmarks—VolanoMark, http://www.volano.com/mark.html.

Java Compiler Technology, http://www.gr.opengroup.org/compiler/index.htm.

Java Grande Forum Benchmark: Links to Other Java Benchmarks, http://www.epcc.ed.ac.uk/research/javagrande/links.html.

Java Grande Forum Benchmark Suite, http://www.epcc.ed.ac.uk/research/javagrande/benchmarking.html.

Java Grande Forum Report: Making Java Work for High-End Computing, JavaGrande Forum Panel, SC98, Nov. 1998, http://www.javagrande.org/reports.htm.

Java Microbenchmarks, http://www.cs.cmu.edu/~jch/java/benchmarks.html.

Jax: Scanner Generator, http://www.meurrens.org/ip-Links/Java/codeEngineering/blackDown/jax.html.

Jell: Parser Generator, http://www.meurrens.org/ip-Links/Java/codeEngineering/blackDown/jell.html.

JHLUnzip—A Zippy Utility, http://www.easynet.it/~jhl/apps/zip/unzip.html.

JHLZip–Another Zippy Utility, http://www. easynet.it/~jhl/apps/zip/zip.html.

JLex: A Lexical Analyzer Generator for Java, http://www.cs.princeton.edu/~appel/modern/java/JLex.

Just-In-Time Compilation and the Microsoft VM for Java, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm/Jit_Structure.htm.

KAZI, I. H. AND LILJA, D. J. 2000. JavaSpMT: A speculative thread pipelining parallelization model for Java programs. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, May, 559–564.

KRALL, A. AND GRAFL, R. 1997. CACAO—A 64 bit Java VM just-in-time compiler. In *Principles & Practice of Parallel Programming (PPoPP) '97 Java Workshop*.

KRALL, A. AND PROBST, M. 1998. Monitors and exceptions: How to implement Java efficiently. In *ACM Workshop on Java for High-Performance Network Computing*.

KRINTZ, C., CALDER, B., AND, HÖLZLE, U. 1999. Reducing transfer delay using Java class file splitting and prefetching. In *OOPSLA '99*, 276–291.

LILJA, DAVID J. 2000. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, New York.

LAUNAY, P. AND PAZAT, J. L. 1997. A Framework for Parallel Programming in Java. IRISA, France, Tech. Rep. 1154, Dec.

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA.

Linpack Benchmark—Java Version, http://www.netlib.org/benchmark/linpackjava/.

MCGHAN, H. AND O'CONNOR M. 1998. PicoJava: A direct execution engine for Java bytecode. *IEEE Computer*, Oct., 22–30.

MEYER, J. AND DOWNING, T. 1997. *Java Virtual Machine*. O'Reilly & Associates, Inc., Sebastopol, CA.

MULLER, G., MOURA, B., BELLARD, F., AND CONSEL, C. 1997. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Conference on Object-Oriented Technologies and Systems (COOTS)*.

MATSUOKA, S., OGAWA, H., SHIMURA, K., KIMURA, Y., HOTTA, K., AND TAKAGI, H. 1998. OpenJIT—A reflective Java JIT compiler. In *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, 16–20.

MOREIRA, J. E., MIDKIFF, S. P., GUPTA, M., ARTIGAS, P. V., SNIR, M., AND LAWRENCE, D. 2000. Java programming for high-performance numerical computing. *IBM System Journal*, vol. 39, no. 1, 21–56.

Microsoft SDK Tools, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm.

O'CONNOR, J. M. AND TREMBLAY, M. 1997. PicoJava-I: The Java virtual machine in hardware. *IEEE Micro*, Mar./Apr., 45–53.

POSTEL, J. B. 1981. Transmission Control Protocol. RFC 791, Sept.

PROEBSTING, T. A. 1992. Simple and efficient BURS table generation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation (PLDI)*, June, 331–340.

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 1997. Toba: Java for applications a way ahead of time (WAT) compiler. In *Conference on Object-Oriented Technologies and Systems (COOTS)*.

Patriot Scientific Corporation. Java on Patriot's PSC1000 Microprocessor, http://www.ptsc.com/PSC1000/java_psc1000.html.

PC Magazine Test Center: JMark 1.01, http:// www8.zdnet.com/pcmag/pclabs/bench/benchjm.htm.

Remote Method Invocation Specification, http://java.sun.com/products/jdk/1.1/docs/guide/rmi-/spec/rmiTOC.doc.html.

SESHADRI, V. 1997. IBM high performance compiler for Java. *AIXpert Magazine*, Sept., http://www.developer.ibm.com/library/aixpert.

SILBERSCHATZ, A. AND GALVIN, P. 1997. *Operating System Concepts*. Addison-Wesley Longman Inc., Reading, MA.

SUGANUMA, T., OLASAWARA, T., TAKEUCHI, M., YASUE, T., KAWAHITO, M., ISHIZAKI, K., KOMATSU, H., AND NAKATANI, T. 2000. Overview of the IBM Java just-in-time compiler, *IBM Systems Journal*, vol. 39, no. 1, 175–193.

SPEC JVM98 Benchmarks, http://www.spec.org/org/jvm98/.

Sun Microsystems—The Source for Java Technology, http://java.sun.com.

Sun Microsystems. MicroJava-701 Processor, http://www.sun.com/microelectronics/microJava-701.

Symantec—Just-In-Time Compiler Performance Analysis, http://www.symantec.com/jit/jit_pa.html.

SYSmark J, http://www.bapco.com/SYSmarkJ.html.

TURLEY, J. 1997. MicroJava Pushes Bytecode Performance—Sun's MicroJava 701 Based on New Generation of PicoJava Core. Microprocessor Rep., vol. 11, no. 15, Nov. 17.

TIP, F. AND SWEENEY, P. 1997. Class hierarchy specialization. In *OOPSLA '97*, 271–285.

TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. 1999. Practical experience with an application extractor for Java. In *OOPSLA '99*, 292–305.

The Java Hotspot Performance Engine Architecture, http://java.sun.com/products/hotspot/-whitepaper.html.

TurboJ Benchmark's Results, http://www.camb.opengroup.org/openitsol/turboj/technical/benchmarks.htm.

TurboJ Java to Native Compiler, http://www.ri.silicomp.fr/adv-dvt/java/turbo/index-b.htm.

UCSD Benchmarks for Java, http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.

UW Cecil/Vortex Project, http://www.cs.washington.edu/research/projects/cecil.

VIJAYKRISHNAN, N., RANGANATHAN, N., AND GADEKARLA, R. 1998. Object-oriented architectural support for a Java processor. In *Proceeding of the 12th European Conference on Object-Oriented Programming (ECOOP)*, 330–354.

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*, vol. 637 of Lecture Notes in Computer Science. Springer-Verlag, New York, Sept. 17–19, 1–42.

Welcome to Harissa, http://www.irisa.fr/compose/harissa/.

Welcome to JMark 2.0, http://www.zdnet.com/zdbop/jmark/jmark.html.

Wilkinson, T., Kaffe v0.10.0—A Free Virtual Machine to Run Java Code, Mar. 1997. Available at http://www.kaffe.org/.

Received December 1998; revised July 2000; accepted August 2000