# Techniques for Obtaining High Performance in Java Programs

Iffat H. Kazi  Berdenia Stanley  David J. Lilja  Amit Verma  Shakti Davis
Department of Electrical and Computer Engineering
Minnesota Supercomputing Institute
University of Minnesota
200 Union St. SE
Minneapolis, MN 55455

**Abstract**

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. 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. 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. Some techniques, including bytecode optimization and executing Java programs in parallel, attempt to improve Java runtime 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 provide information about the various Java benchmarks that are being used by the Java community for performance evaluation of the different techniques. Finally, we conclude with a discussion comparing 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

**Keywords and Phrases:** Java, Java Virtual Machine, Interpreters, Just-In-Time compilers, Direct Compilers, Bytecode-to-source Translators

# 1 Introduction

The Java programming language evolved out of a research project started by Sun Microsystems in 1990 [3, 10]. It is one of the most exciting technical developments in recent years since Java's "write-once, run anywhere" philosophy captures so much of what developers have been looking for in a programming language in terms of application portability, robustness, and security. Java is a combination of four different styles of programming [11]. First, it captures the flavor of object-oriented programming. Second, the complexity of numerical programming has decreased when compared to other programming languages such as C. For example, there is only one way to say "n=n+1", unlike in C in which both "n++" and "n+=1" mean the same thing. Third, Java incorporates many of the systems programming concepts from C. And, finally, it is architecture independent.

Java has been defined as [9]:

> "A simple, object-oriented, distributed, interpreted, robust, secure, architecturally neutral, portable, high-performance, multithreaded, and dynamic language."

It is referred to as **simple** because many constructs already look familiar to programmers due to its similarities with other languages. The **object-oriented** characteristic comes from the idea that *methods* are used to manipulate data. No procedural thinking is used. Instead, a *class* is comprised of both data and methods. Classes exist in a hierarchy such that a subclass inherits behavior from its superclass. Java is said to be a **distributed** language because it directly supports network connectivity. It is an **interpreted** language in which an interpreter program executes the compiled *bytecodes* generated by the Java compiler. In fact, the bytecode defines the instruction set for a (possibly virtual) Java processor.

Java was designed to implement highly reliable or **robust** programs. For example, unlike C, no pointers are used in Java programs. Thus, the possibility of accidentally overwriting memory is eliminated. Additionally, automatic garbage collection is provided to eliminate problems with memory allocation and deallocation. **Security** mechanisms are used in Java to protect the user's system from malicious programs downloaded in a networked environment. Techniques such as mapping application program memory references to real

addresses at runtime and using special processes to verify downloaded code ensure that the Java language restrictions are enforced. The concept of **architectural neutrality** allows applications to run on any system that can run a *Java Virtual Machine*, which is the program or the device that executes the machine independent bytecodes produced by the Java compiler. This architectural neutrality makes Java programs **portable**.

The Java programming language is **multithreaded** since it handles multiple threads of execution. It is **dynamic** due to its ability to handle a changing environment, including loading classes as needed at runtime.

Even though the above description of the Java programming language claims that it is high performance, the cost of its flexibility is its relatively slow performance due to interpretation. While advances with *Just-In-Time compilers* are making progress towards improving its performance, existing Java execution techniques do not 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 execution techniques that are currently being used with the Java programming language. Section 2 describes the basic concepts behind the Java Virtual Machine (JVM) interpreter and the Java class file, which contains the Java bytecodes. Section 3 discusses the advantages and disadvantages of the different software-based Java execution techniques, including bytecode optimization and parallel execution of Java. Java processors that implement the JVM directly in silicon are discussed in Section 4. 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 *Java Virtual Machine* which actually executes the program.

## 2.1 Java Virtual Machine

The *Java Virtual Machine*, often referred to as the *JVM*, interprets the Java program's bytecodes [21, 18]. The interpreter is said to be *virtual* since, in general, the JVM 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 Java Virtual Machine provides the interface between compiled Java programs and any target hardware platform.

The JVM actually 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 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 Java Virtual Machine'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* 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

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

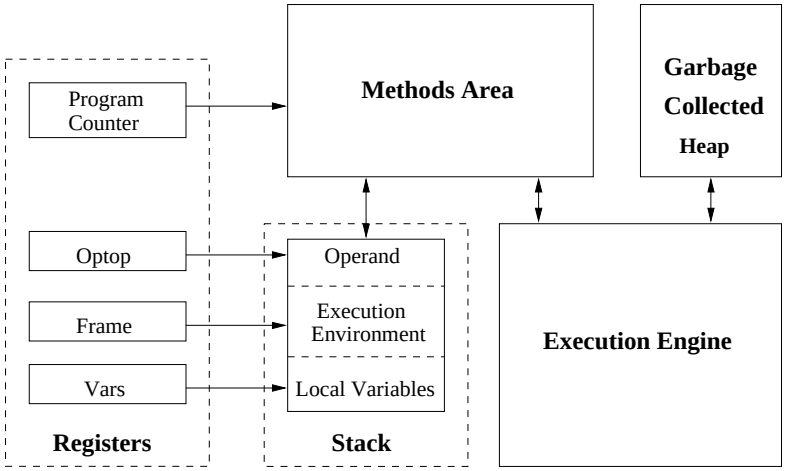


Figure 1: Basic components of the Java Virtual Machine.

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.

The JVM is a stack-based machine where all operations on data 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 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 zeros. 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 2.3.

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 on silicon. The execution engine interacts with the method area to retrieve the bytecodes for execution. Various implementations of the Java execution engine are described in subsequent sections.

## 2.2 Class File Details

Java source code is initially converted into the bytecode format using a Java compiler. A number of Java compilers have been developed, including Sun Microsystems' *javac* [W1] and Microsoft's *jvc* [W2]. These compilers take the *.java* source files as input and produce *.class* files that contain the corresponding Java bytecodes. These bytecodes are platform-independent and can be run on any system that can execute a JVM. A *class* in the Java source code defines *fields* (data variables) and *methods* (procedures or functions) common to all objects of the same type. A *class file* contains all the information about a particular class that is needed by the JVM. It consists of a number of components that must appear in a designated order. No spaces exist between components in an attempt to minimize the file size.

Figure 2 shows the structure of the class file generated by a Java compiler. The first four bytes of every class file comprise the *magic number*, 0xCAFEBABE, that identifies a file as a Java class file. This *magic number* was defined by the file format designers of the original Java team. The next four bytes refer to the *major* and *minor* version numbers, which determine the version of the class file format. Since each virtual machine has a maximum version that it can load and execute, the version number is used to determine

whether the given class file can be executed by the running JVM.

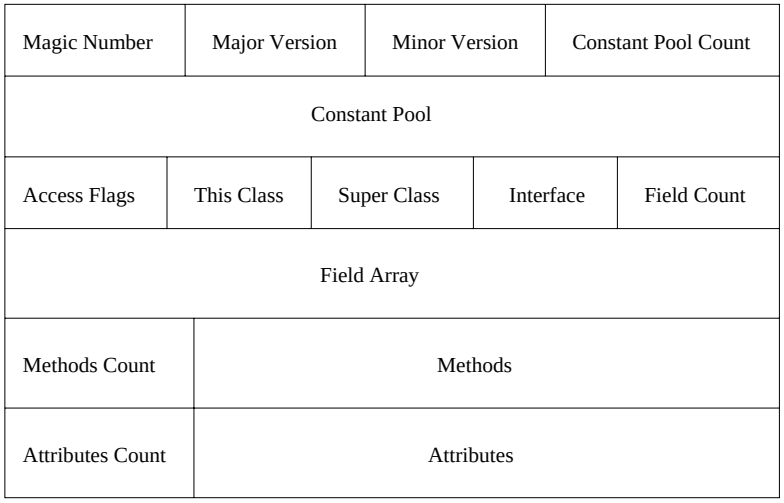


Figure 2: Java class file structure.

The *constant pool* appears next in the class file. It is an array of variable-length elements that are the constants used in the class. Each constant is stored in one element of the array. Within the class file, constants are identified by an integer index into the array. The size of the constant pool array precedes the actual array in the class file which enables the virtual machine to determine the number of constants associated with the class file being loaded.

The two bytes immediately following the constant pool are called the *access flags*. These flags identify whether a *class* or an *interface* is being defined. An *interface* is a prototype for a class that includes only the declarations of the methods without the implementation details. The access flags are followed by two bytes that make up the *this class* component and two bytes that make up the *super class* component. Both of these components are used to index into the constant pool. The *interface* component uses two bytes to define the number of interfaces defined in the class file. The *field* component appears next in the class file. A *field* is defined to be a class variable. The *field count* provides the number of fields in the class. An array immediately following the field count contains a variable-length structure for each field present. Each of these structures contain field information such as the name, the type and so forth.

Following the field component is the *methods* component. This component begins with a two-byte count that indicates the number of methods that are explicitly defined in the class. All methods in the class immediately follow this count. The class file ends with general information about the class, referred to as the *attributes* component. Two bytes are used to identify the number of attributes with the actual attributes found immediately following the count. The *source code* attribute identifies the Java source file compiled to produce the specific class file, for instance.

## 2.3 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 Java virtual machine specification [10] 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 *non-copying implicit collection* [28]. 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

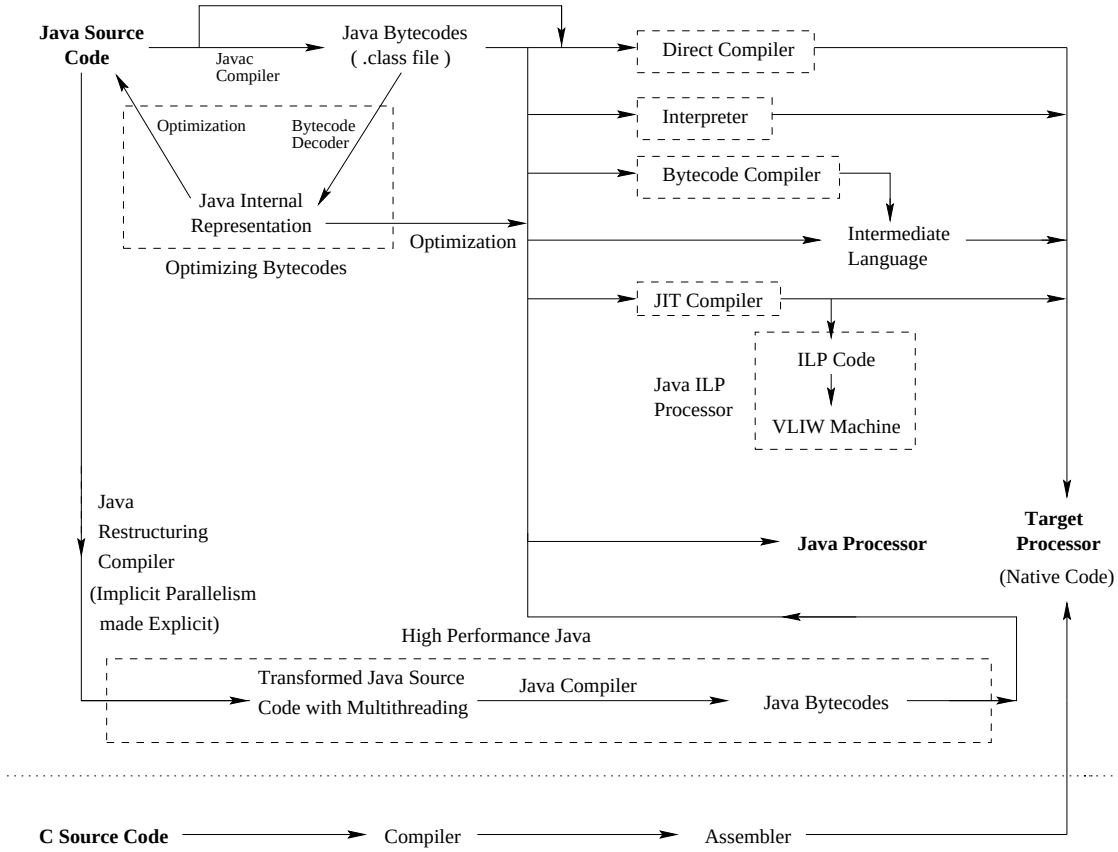


Figure 3: Alternatives for executing Java compared to a typical C programming language compilation process.

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 3.

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, a number of optimization techniques have been developed to enhance the performance of Java programs, including bytecode optimization and executing Java programs in parallel. These optimization techniques are presented in Section 3.3.

## 3.1 Java Interpreters

Java interpreters are the original and, perhaps, most common means for running 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 JVM reads and executes each of the bytecodes in order. An interpreter has several advantages over a traditional compiled execution. Interpretation is very simple, 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. Since the translations to native code are not cached, code repeated in a loop, for instance, must be translated over and over again while the Java program is running. This translation process can be very time-consuming.

There are several existing Java interpreters. The Sun Java Developers Kit (JDK) [W1] from Sun Microsystems is used to develop Java applications and applets that will run in all Java-enabled web browsers. A Java applet is a mini-application that can be run inside a web browser or applet viewer. It is not a stand-alone program, and, therefore, cannot be run directly using a Java command. 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 [W4]. The SDK contains

the Microsoft Win32 Virtual Machine for Java (Microsoft VM), classes, APIs, and so forth for the i386 and ALPHA platforms.

## 3.2 Java Compilers

Another technique to execute Java programs is with a compiler. Like traditional high-level language compilers, a *direct Java compiler* starts with an application's Java source code and translates it directly into the machine language of the target processor. Other Java compilers, such as the *Just-In-Time compilers*, are more like traditional assemblers in that they take the Java bytecode as their starting point. Several of these different types of compilers are described in the following subsections.

### 3.2.1 Just-In-Time Compilers

As shown in Figure 3, a *Just-In-Time (JIT) compiler* translates Java bytecodes into native machine instructions. This translation is performed on-the-fly at runtime so that methods are compiled on an "as needed" basis. This means that the *.class* files are not all compiled before the program is loaded and executed. Instead, the JIT compiler generates native code sequences for the target processor as the bytecodes are encountered during execution. These sequences are cached to eliminate the repeated translation of the same sequence of bytecodes. This approach reduces the number of translations to one, whereas an interpreter must continually translate each time the bytecode sequence is encountered. In programs that have a large number of loops or use recursion extensively, the total required translation time is drastically reduced compared to an interpreter.

In a typical JIT compiler, the *bytecode importer* converts the instruction bytecodes into an internal representation [W5]. This internal representation can be optimized and then passed to the code generator, where the native code is created. In other words, the JIT compiler replaces the bytecodes with native machine code that performs the same function.

The increase in speed over interpreters provided by JIT compilers may come with some potential disadvantages. For instance, code size often increases during compilation which results in higher memory

consumption. Furthermore, the global program information required by some compiler optimization techniques is not available to JIT compilers. Instead, since compilation occurs on demand for one class or one method at a time, only local optimizations can be performed. Another disadvantage of the JIT process is the introduction of a start-up lag time that occurs while the first methods are being compiled. Also, the native code generated is based on a specific target processor, which severely limits the portability of this type of compiler. Furthermore, since the native codes generated by the JIT compiler are not stored in external files for future runs, the bytecodes must be recompiled every time the program is executed. The advantage, however, is that significant performance improvement has been achieved using JIT compilers when compared to interpreters. This improvement has been attributed primarily to the reduction in translation time.

The goal of JIT compilation is to improve the performance of Java programs. However, the total execution time perceived by the user is the compilation time plus the time actually required to execute the compiled program. Thus, there is tradeoff between improving execution time performance by applying extensive compile-time optimizations and the extra time required by the compiler to perform these optimizations.

The first implementations of JIT compilers were developed for Internet browsers, such as Netscape's JIT and Microsoft's Internet Explorer JIT. The Netscape JIT was developed by Borland and has been integrated into the Netscape Navigator for computers running Windows 95 and NT [W6]. This JIT compiler translates Java bytecodes into Intel machine code. The Microsoft Internet Explorer offers JIT compilation [W7] for the Windows 95, NT, Macintosh, and UNIX environments. Additional JIT compilers have been developed, such as Symantec Cafe's JIT [W8], where the performance of applets was a key issue in the development process, and Silicon Graphics' JIT compiler for the MIPS processor [W9]. The *Kaffe* JIT compiler [W10] uses a machine-independent front-end to translate bytecodes into an intermediate representation known as *Kaffe IR*. The back-end then translates Kaffe IR into the designated machine language, including the i386, SPARC, M68K, ALPHA, PowerPC, and MIPS processors.

The *SunSoft* JIT compiler is available for both the Intel and SPARC processors. It has been integrated into a complete Java development tool kit known as the Java Workshop [W11]. Digital Equipment Corporation's JIT compiler was introduced as a part of the JDK v1.1.1 [W12] for the Digital Unix operating system.

Another version of the JDK with a JIT compiler was released as part of the OpenVMS Alpha operating system. IBM has implemented a version of Sun's Java Development Kit with JIT compilation technology for the OS/2 Warp and AIX platforms [W13, W14], as well as the AS/400.

*CACAO* is another JIT compiler that translates Java bytecodes into native code for the ALPHA processor [17]. Since the architecture of the ALPHA processor is quite different from the stack architecture of the Java virtual machine, CACAO has been specially developed to handle the 64-bit architecture of the ALPHA processor by transforming Java bytecodes into a register-oriented intermediate code.

*Guava* is a JIT compiler for the SPARC platform that incorporates threads to allow the use of multiple CPUs on Symmetric Multiprocessor (SMP) machines [W15]. *Footprint Aware Just-In-Time for Appliances* (FAJITA) is a modified JIT compiler that targets embedded systems [W16]. FAJITA transforms bytecodes into native code just like any other JIT compiler, but it saves the compiled code for subsequent usage. As such, it is not integrated into the virtual machine. FAJITA generates native code with a primary goal of reducing memory requirements to fit within the constraints of embedded systems.

### 3.2.2 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 3). The 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. Also, direct compilers can view the entire program to perform global optimizations in contrast to a JIT compiler that can optimize only locally. As a result, aggressive optimization techniques can be used to improve the performance of the native code generated. Another advantage of direct compilers is faster compilation since the intermediate translation of Java source code into bytecode can be eliminated. Direct compilers also provide an advantage to applications whose execution time exceeds the translation time since the native code is run directly on the target processor. However, direct compilation results in the 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.

*Supercede* developed the first native Java compiler for the Intel/Win32 platform [W17]. The *Native Executable Translation* (NET) compiler uses state-of-the-art compiler analysis in the translation process [13]. Issues such as mapping the stack execution model to a register-oriented execution model and efficient memory organization were considered in the development of the NET compiler. With the goal of yielding performance close to that achieved by compiled C code, the developers of the *Caffeine* native code translator [14] addressed issues such as stack to register mapping, mapping the bytecode memory organization to the native architecture, and exception handling translation. The *High Performance Compiler for Java* (HPCJ) [W18, W19] was developed by IBM researchers to convert the Java bytecode directly into native code for the target processor. Extensive optimization techniques are used to obtain the highest possible performance. HPCJ has been targeted for the AIX, OS/2, Windows 95, and Windows NT platforms.

### 3.2.3 Bytecode-to-Source Translators

*Bytecode-to-source* translators generate an intermediate high-level language, such as C, from the Java bytecodes (see Figure 3). A standard compiler is then used to generate executable machine code. This process completely eliminates the overhead of interpretation or JIT compilation every time a program is executed. These translators allow more flexibility than direct Java to machine code compilers since portability is maintained. Choosing a high-level language such as C as an intermediate language allows the use of existing compiler technology, which is useful since C compilers that incorporate extensive optimization techniques are available for almost all platforms.

*Toba* [24] is an example of a bytecode-to-source translator that converts Java class files directly into C code, which is then compiled into machine code. Toba is referred to as a Way-Ahead-of-Time (WAT) compiler since it compiles during program development in contrast to a Just-In-Time (JIT) compiler that performs the necessary translation immediately before the execution of each class. *J2C* is a bytecode to C translator for the i386 and SPARC platforms [W20] that has several restrictions, such as no support for dynamic class loading, network resources, or threads. Furthermore, it performs no optimizations when

generating the C code.

*TurboJ* is a Java bytecode-to-source translator [W21, W22] that also uses C as an intermediate representation. It works separately from the Java runtime system while implementing optimizations similar to traditional compilers. Its goal is to generate highly optimized code that yields the best possible performance. Since TurboJ generates C code, standard optimizing C compilers can be used for the final code generation.

*Vortex* is an optimizing compiler that supports general object-oriented languages [W23, 7]. The Java, C++, Cecil, and Modula-3 languages are all translated into a common intermediate language (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 dead code elimination, method inlining [1], and those specifically targeting object-oriented languages. Once optimization is complete, either C code or assembly code can be generated.

The *Java Hot Set Compiling Technology*, which is being developed by the Open Research Institute [W24, W25], operates on bytecodes with a combination of the FAJITA and TurboJ compilers. The goal of this project is to determine the best trade-off between cost requirements and performance. Hot Set minimizes the memory required by compiling only the specific code necessary for an application while excluding the standard Java features not used by the application. This technology was developed for embedded systems, network computers, and client-server systems.

The *Java Open Language Toolkit* (JOLT) project was created to produce a "redistributable implementation" of Sun Microsystems Java environment [W26]. The idea was to provide a Java toolkit to users without Sun's license restriction. This "clone" of Sun's Java will meet the requirements of Sun's validation suite. JOLT will include a Java compiler called guavac, which is a new compiler written in C++, and a Kaffe bytecode interpreter, class library, etc. Linux/i386 is the initial targeted platform for this project.

*Harissa* is another Java environment that includes both a bytecode translator and an interpreter [22, W27] for SunOS, Solaris, Linux, and DEC Alpha. The bytecode translator, referred to as *Hac*, translates Java bytecodes to C code. Since existing compilers can be used to perform the final optimizations, Harissa focuses only on optimizing stack evaluations and method calls. The interpreter in the Harissa environment

has been integrated into the runtime library to allow code to be dynamically loaded into previously compiled applications. Since data structures between the compiled code and interpreter are compatible, Harissa provides an environment that cleanly allows the mixing of bytecodes and compiled code.

*HotSpot*, developed by Sun, is a combination of the best features of an interpreter and a JIT compiler. HotSpot includes a "dynamic compiler", a virtual machine, and a profiler. The dynamic compiler includes an interpreter and a JIT compiler. Initially, the interpreter executes the Java bytecodes while the profiler records the method execution times. The profiler then predicts the method optimization time. Once a method's runtime reaches a predefined percentage of the profiler's prediction of the required optimization time, the method is compiled and optimized by HotSpot to produce native machine code that is stored and reused for subsequent calls.

HotSpot's dynamic compilation provides several advantages. One is that exceptions can be ignored by the optimizer since it is guaranteed that exceptions will be handled by the interpreter. This fallback exception-processing strategy allows better optimized code to be produced since all possible cases do not have to be considered. Also, method inlining reduces the overhead of method calls to improve performance. The disadvantage, however, is a corresponding increase in program size. Depending upon the size of the method, its execution time, and the number of times it is invoked, inlining may not always be the best choice.

## 3.3 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, but this high performance comes at the expense of loss of portability. JIT compilers, on the other hand, maintain portability but cannot obtain much performance improvement as only limited optimizations can be done. A number of techniques have been developed that attempt to improve Java performance by optimizing Java source code or bytecodes so that portability is not lost. In this section, we describe several high performance Java execution techniques.

### 3.3.1 Bytecode Optimization

One technique for improving execution time performance of Java programs is to optimize the Java bytecodes of a program. The *Briki compiler* was developed to investigate the potential benefits of high-level optimizations for Java programs [5, 6]. Java bytecodes do not provide enough high-level program structure information to apply standard optimization techniques, however. Consequently, the front-end of the offline mode of the Briki compiler [6] recovers the full 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*. High-performance optimization techniques, such as array and loop transformations, can be efficiently applied to this JavaIR. The optimized JavaIR then can be transformed back to Java source code which is then input to the Java compiler. However, converting Java bytecodes to JavaIR is very expensive requiring an order of magnitude more time than the time to perform the optimizations.

In the current implementation of Briki [5], the same optimizations are performed while recovering only as much structure as needed and using faster analysis techniques than those used in traditional compilers. Briki is integrated with the Kaffe JIT compiler [W10]. It uses the Kaffe front-end to generate its intermediate representation (IR) from the Java bytecodes. The compiler analyzes and transforms the Kaffe IR into an optimized IR which the Kaffe back-end then translates into the native code.

One optimization in Briki attempts to improve memory locality by data remapping. Since the JVM is stack-based, it has no concept of an address. Instead, data layout optimization techniques must be used to change the layout of data structures to improve memory utilization when executing the stack-based Java programs on a traditional register-to-register processor architecture. One technique attempts to improve the cache locality by changing the array layout so that more array elements lie in the cache. Another technique positions fields that are likely to be accessed together in consecutive memory locations. This optimization can increase the cache hit ratio since the consecutive memory locations are likely to be in the same cache line. Other approaches, such as remapping arrays to be rectangular and changing the evaluation order of a remapped array reference, have also been considered.

Preemptive Solutions has developed a bytecode optimizer called *DashO Pro* [W28]. DashO 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. All of these techniques aid in producing smaller and faster Java bytecode files.

### 3.3.2 Parallel Java

The *Parallel Java* project is based on the Converse interoperability framework [15]. This framework allows parallel applications to be developed with its individual parallel modules written using different paradigms and languages. Parallel Java extends standard Java to allow parallel execution of Java applications. It does not require any modification to the Java compiler or the JVM. Hence, any standard Java compiler and JVM can be used to execute parallel applications written in standard Java with calls to a special runtime library.

In Parallel Java, the programmer must define a main class, an instance of which is automatically created on processor 0. After the static method *main* of this main class exits, the system invokes the scheduler and waits for messages in the scheduler loop. Other processors go directly into the scheduler loop at the beginning. *Remote classes* are defined that are similar to the regular Java classes except that remote objects are created and accessed through the use of *proxies*. Certain *entry methods* can be accessed remotely in the remote class. A *proxy* class provides access to the remote objects through the implementation of similar entry methods. In addition to these classes, a *message class* must be defined by the programmer for every type of message that an entry method can receive. All of the parallel classes, messages, and entry methods must be registered with the runtime system so that it can determine which method to invoke, how to unpack a message, and the object for which a method is intended.

A Parallel Java application can be run on a network of workstations using the Converse utility called *conv-host* and a Parallel Java virtual machine (*pjava*). A Parallel Java virtual machine is a C program that uses Converse libraries and embeds the JVM using the Java Invocation API. The *conv-host* utility reads the

IP address of the individual workstations from a local file to spawn the parallel Java virtual machine on those workstations. After *pjava* is started, *conv-host* passes it the main class and other parameters. The *pjava* program loads the Parallel Java runtime class and registers the remote objects using the Java Invocation API. It then invokes the *main* method of the main class on processor 0 followed by the scheduler.

### 3.3.3 High Performance Java

In the *High Performance Java* system [4], the *javar* restructuring compiler makes some of the parallelism implicit in Java programs explicit with the language's multithreading mechanism. The emphasis in High Performance Java is automatically exploiting implicit parallelism in loops and multi-way recursive methods. The parallel loops can be identified either by means of data dependence analysis [29] or by a programmer's explicit annotations. Parallelism in multi-way recursive methods must be identified by explicit annotation since parallelism is very difficult to detect automatically in these types of methods.

The *javar* source-to-source compiler transforms the input program into a form that uses Java's multithreading to make all implicit parallelism explicit. The transformed program is then compiled into bytecodes using any Java compiler. Since parallelism is expressed in Java itself, the transformed program remains portable and speedup can be obtained on any platform on which the Java bytecode interpreter supports true parallel threads. *Fork/join*-like parallelism is used to execute parallel loops on shared-address space architectures. This type of parallelism starts a number of threads, each executing the different iterations of the loop in parallel, for instance.

Implementation of parallel loops in Java is based on a class hierarchy. The first layer in this hierarchy is the Java *Thread* class which is provided in the *java.lang* package. The second layer, which is also independent of the source program, is specially developed to run parallel loops. This layer consists of the abstract class *LoopWorker*, the *LoopPool* class, which is derived from the *Schedules* interface, and the *RandomSync* class.

The *LoopWorker* class provides an abstraction derived from the *Thread* class that executes several iterations of a parallel loop. A new class, *LoopWorker_x*, forms the third layer of the class hierarchy. It is constructed explicitly by the restructuring compiler for each parallel loop in the program and added to the

Java program. These classes are derived from the *LoopWorker* class. When a parallel loop is executed, the loop-workers compete for work by accessing a shared pool of iterations. The structure of the shared pool is defined by the class *LoopPool*. The *Schedules* interface class defines various scheduling policies for the shared pool of iterations. Iterations to be executed as one block of work by a loop-worker are identified using the instance variables *low*, *high*, and *stride*, where *low* and *high* represent a consecutive range of iterations to be executed, and *stride* is the step between consecutive iterations. The instance variable *sync* within each loop-worker is used for synchronization. The variable *sync* is an object of the *RandomSync* class that defines synchronization variables for random synchronization of DOACROSS loops.

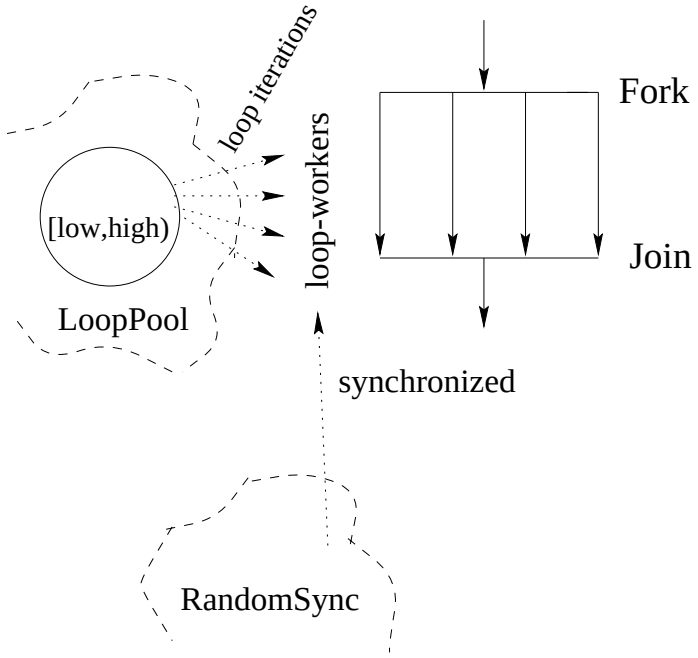


Figure 4: Execution of a parallel loop in the High Performance Java system.

Initially, new instantiations of a work pool and a required number of synchronization variables are obtained from the *LoopPool* class and the *RandomSync* class, respectively. These work pool and synchronization objects are then made available to all the loop-workers and a fork operation is performed. When all iterations have been executed, each loop-worker performs a *join* operation to wait for all threads to finish. Since a

pool is shared amongst all loop workers of a particular parallel loop, method synchronization allows mutual exclusion for updating the shared data structure. Figure 4 shows the parallel execution of a loop using this model.

A direct approach for exploiting implicit parallelism in a parallel $n$-way recursive method is to allow all but one of the recursive method invocations to be assigned to other threads by a running thread. The easiest way to assign work to a limited number of processors is through a static allocation where the running threads assign method invocations to other threads only in the top levels of the method invocation tree. The assumption is that most multi-way recursive methods try to keep the method invocation tree reasonably balanced. If $m$ parallel multi-way methods are contained in a Java program, then $m$ classes are constructed and added to the transformed Java program. These new classes are referred to as *TreeWorker_x*. Instance variables are used to transfer the result of a method invocation and to record the current depth in the method invocation tree. The total number of invocations allowed is limited by the compiler constant CUT_DEPTH.

# 4 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. While a JIT compiler can provide significant speedups over an interpreter, it introduces additional compilation time and requires 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 has to 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 [23]. In particular, there are certain common characteristics of Java programs that are different than 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 architectural features in 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 5 shows the dynamic frequencies of the various JVM instruction categories measured in Java benchmark programs *LinPack*, *CaffeineMark*, *Dhrystone*, *Symantec*, *JMark2.0*, and *Java-World* (benchmark program details are provided in Section 5). The most frequent instructions are local variable loads (about 30-47% of the total instructions), 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 will yield better performance. Method calls and returns are also quite common in Java programs. As seen in Figure 5, they constitute about 7% of the total instructions executed. Hence, optimizing the method call and return process is expected to have a large impact on the performance of Java codes. Since method calls occur through the stack, a Java processor 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 want to enter synchronized critical sections of code to provide mutually exclusive access to shared objects. Thus, a Java processor needs to 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* [23, 20] is a configurable processor core that supports the JVM specification. It includes

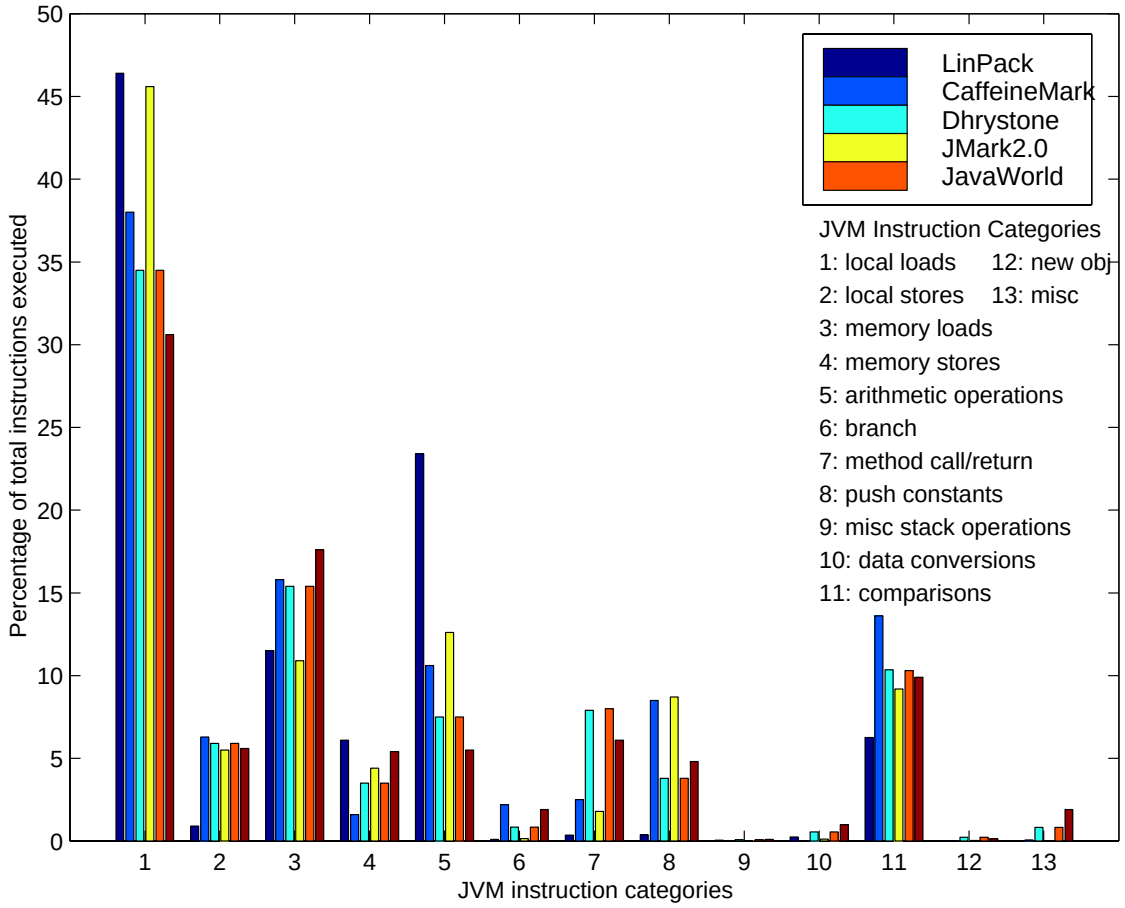


Figure 5: Dynamic execution frequency of JVM instructions.

a RISC-style pipeline executing the JVM instruction set. However, only those instructions that directly improve Java 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.

The picoJava-I core implements a hardware stack using a 64-entry on-chip cache. Java bytecode execution is accelerated by *folding* operations to take advantage of the random, single-cycle access to the stack cache. 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. 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 [25], which is the successor to the picoJava-I processor, augments the byte code 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. 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 picoJava-I. Finally, the *folding* operation is extended to include two local variable accesses.

Another Java processor is the *Sun microJava 701* microprocessor [W29]. It is based on the picoJava-2 core and is supported by a complete set of software and hardware development tools. Rockwell's Java processor *JEM1* [W31] is also based on the picoJava-I core. The *Patriot PSC1000* microprocessor [W30] 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 [27] 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

- *hybrid cache*, *hybrid polymorphic cache*, and *two-level hybrid cache* - have been proposed to efficiently implement virtual method invocations. The processor uses *extended folding* operations as 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 [8] incrementally compiles Java bytecodes into a Very Long Instruction Word (VLIW) parallel 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 incremental 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 for applications written in any other language, unless a C to Java compiler is available, for instance. 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.

# 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. 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 currently exists in this research area is that no benchmarks have been identified as the standard set to be used for Java performance analysis [14]. Instead, researchers currently use 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 [W32, W33]. The following programs fall into the category of application benchmarks:

- *JavaLex* [W52] - a lexical analyzer generator.
- *Javac* [W1] - Sun's Java source code-to-bytecode compiler. As one of the longer Java programs available, it has become quite popular as a benchmark program.
- *JavaCup* [W53] - a parser generator for Java.
- *JHLZip* [W54] - combines multiple files into one archive file with no compression.
- *JHLUnzip* [W55] - extracts multiple files from JHLZip archives.

In addition to these benchmarks, the Open Systems Group (OSG) of the Standard Performance Evaluation Corporation (SPEC) [W34] is developing a suite of Java benchmarks for processor and system analysis. The *Linpack* benchmark is a popular Fortran benchmark that has been translated into C and is now available in Java [W37]. This numerically intensive benchmark measures the floating point performance of a Java system by solving a dense system of linear equations, Ax=b.

A series of microbenchmark tests, known as *CaffeineMark* [W38], 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 [2], there is a question as to whether this benchmark makes reliable measurements of performance. The *Embedded CaffeineMark benchmark* [W38] 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 [W39, W40], 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. This JMark is one of the few benchmarks that attempts to evaluate multithreading performance. *SYSmark J* is a new benchmark developed by the Business Applications Performance Corporation (BAPCo) [W41] and is claimed to be the first benchmark based on Java applications. SYSmark J consists of four applications: *JPhotoWorks*, 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.

*VolanoMark* [W42] 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.

Other available benchmarks include *JByte*, which is a Java version of the *BYTEmark* benchmark available from BYTE magazine [W43]; the *Symantec* benchmark, which incorporates tests such as *sort* and *sieve* [W8, W43]; the *Dhrystone* CPU benchmark [W44, W45]; *Jell*, a parser generator [W46]; *Jax*, a scanner generator [W47]; *Jas*, a bytecode assembler [W48]; and *EspressoGrinder*, a Java source program-to-bytecode translator [W49]. Additional benchmark references are available on the Java Benchmarking [W50] and Java Benchmark References [W51] web sites.

# 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. While incomplete and difficult to compare directly, these performance results do provide a limited means of comparing the alternative techniques to determine which ones provide an improvement in performance and to what degree. Note that some of these results are gathered from self-published web references that are sometimes more interested in selling a product than

providing scientifically reproducible results.

Table 1 summarizes the relative performance of the various techniques based on reported results [2]. 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 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. Moreover, the performance evaluations are based on a variety of different benchmarks making it difficult to directly compare the different execution techniques.

At present, the performance of Java interpreters is roughly 10 to 50 times slower than compiled C/C++ performance [W24]. JIT compilers typically improve the performance by an average of 10-20 times compared to the interpreters. The Microsoft JIT has been reported to achieve approximately 23 percent of "typical" compiled C/C++ performance. That is, the JIT executes Java code approximately 4 times slower than compiled C/C++ code. When compared to the Sun interpreter, however, an average speedup of 5.6 was obtained [13]. Since these performance results for the Microsoft JIT are part of the performance evaluation of the NET direct compiler, the benchmarks used in the evaluation are listed later while discussing NET performance.

Another performance analysis showed a 30 percent improvement of the Microsoft JIT over the Netscape JIT [W7]. According to [W8], the Symantec Cafe' JIT is up to 20 times faster than the Sun JDK interpreter. This performance analysis used the benchmarks *Sieve, Hanoi, Dhrystone, Fibonacci, Array, Tree, Bubble Sort,* and *Quick Sort.* Since these are very small programs, however, it is not clear whether this performance difference will remain the same on real, substantial applications.

[2] In this table, the superscript "*I*" is used to identify results reported from work published in peer-reviewed papers, while the superscript "*II*" defines results based on self-published reports available from various web sites.

| | | Interpreters | JIT Compilers | C/C++ |
|---|---|---|---|---|
| Interpreters | Sun JDK | | | 10x - 50x slower$^{II}$ |
| JIT Compilers | Microsoft | 5.6x faster$^{I}$ | | 4x slower$^{I}$ |
| | Symantec Cafe | 20x faster$^{II}$ | | |
| | Kaffe | 10x faster$^{I}$ | | |
| | Guava | 2.5x faster$^{I}$ | | |
| | CACAO | 2x - 85x faster$^{I}$ | | 1.66x slower$^{I}$ |
| Direct Compilers | NET | 17.3x faster$^{I}$ | 3x faster | 1.2x - 3.6x slower$^{I}$ |
| | Caffeine | 20x faster$^{I}$ | 4.7x faster | 1.5x slower$^{I}$ |
| | HPCJ | 17x faster$^{II}$ | 13x faster$^{II}$ | |
| Bytecode Translators | Toba | 3x - 10x faster$^{I}$ | 1.5x - 2.2x faster$^{I}$ | |
| | TurboJ | 20x - 140x faster$^{II}$ | 2x - 10x faster$^{II}$ | |
| Java Processors | picoJava-I | 15x - 20x faster$^{I}$ | 5x faster$^{I}$ | |

Table 1: Reported relative performance of different Java execution techniques and compiled C/C++.

Kaffe performs up to ten times faster than the Sun interpreter for the benchmarks *Sieve, Linpack,* and *addition* [17], while Guava is about 2.5 times faster than the Sun interpreter when evaluated using the benchmarks *JavaLex, javac, espreso, Toba,* and *JavaCup* [17].

Java programs run 2 to 85 times faster using the CACAO system when compared to the Sun JDK interpreter [17]. Note that these results were obtained with an implementation of CACAO that provides support for garbage collection using a conservative mark-and-sweep algorithm, and with array bounds checking and precise floating-point exceptions disabled. CACAO performs up to 7 times faster than the Kaffe JIT compiler. When compared to compiled C programs optimized at the highest level, though, it performs about 1.66 times slower. However, the CACAO results in Table 1 show some inconsistencies when compared to the results of the Microsoft JIT. If we consider the results for the Microsoft JIT to be valid, the CACAO JIT being 85 times faster than the interpreter should make it faster than compiled C/C++ codes instead of

being 1-66 times slower. On the other hand, if we take the CACAO results to be correct, the Microsoft JIT should be substantially more than 4 times slower than C/C++ code. The inconsistencies can be attributed in part to the benchmarks used in CACAO - *Sieve, addition*, and *Linpack*. The *addition* benchmark is a loop that performs only simple additions. The speedup of 85 over the JDK interpreter was achieved using this benchmark.

Performance evaluation results of a number of direct compilers show their improved performance over interpreters and JIT compilers. The NET compiler, for instance, achieves a speedup of up to 45.6 when compared to the Sun interpreter [13] with an average speedup of about 17.3. Compared to the Microsoft JIT, NET is about 3 times faster on average, and currently achieves 28 to 83 percent of the performance of compiled C/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*; the Java versions of the SPEC benchmarks *026.compress, 099.go, 132.ijpeg*, and *129.compress*; C/C++ codes *Sieve, Linpack, Pi, othello,* and *cup*; and the Java codes *JBYTEmark* and *javac*.

Preliminary results indicate that the Caffeine compiler yields, on average, 68 percent of the speed of compiled C code [14]. It runs 4.7 times faster than the Symantec Cafe JIT compiler and 20 times faster than the Sun interpreter. The reported results for Caffeine are based on an implementation that does not include garbage collection, exception handling, or thread support, however. The benchmarks used are *cmp, grep, wc, Pi, Sieve,* and *compress*.

The performance results obtained for the High Performance Compiler for Java (HPCJ) show a speedup of up to 17 times faster than the AIX JVM interpreter and 13 times faster than the IBM JIT compiler [W18]. The HPCJ results include the effects of a conservative garbage collection mechanism. Since it uses the same back-end as IBM's compiler for C/C++, Fortran and other languages, it is able to utilize a wide set of language-independent optimizations. HPCJ reduces the run-time checking overhead by performing a simple bytecode stack simulation. 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*.

The Caffeine and HPCJ results in Table 1 are inconsistent if we consider the performance relative to the JIT compilers. Both Caffeine and HPCJ perform almost equally well when compared to the interpreter. However, Caffeine seems to be much slower than HPCJ when compared to the JIT compilers. Since the comparisons were made using different interpreters and JIT compilers (e.g. the Symantec JIT for Caffeine versus the IBM JIT for HPCJ), we cannot expect any direct correspondence between the reported results, though.

Codes generated by the bytecode compiler Toba when using a conservative garbage collector, thread package and exception handling, are shown to run 3-10 times faster than the Sun JDK interpreter and 1.5-2.2 times faster than the Guava JIT compiler [24]. The results are based on the benchmarks *JavaLex, JavaCup, javac, espresso, Toba, JHLZip,* and *JHLUnzip*. The speedup of Toba over the interpreter is due to the elimination of the runtime overhead of interpretation and dynamic loading. It performs better than a JIT compiler such as Guava since there is no overhead for runtime code generation. In addition, its bytecode to C translator allows more aggressive optimization of the resulting C code.

According to [W22], TurboJ performs 20 to 140 times faster than the JDK interpreter and 2 to 10 times faster than a JIT compiler. TurboJ incurs a lower start-up time penalty than a JIT compiler since the code is pre-packaged in dynamically loadable libraries. This feature accounts for its better performance over a JIT compiler. TurboJ was evaluated using the *Embedded Caffeine, Linpack, JByte, Symantec, Dhrystone,* and *UCSD* benchmarks. The FAJITA JIT compiler does not perform as well as the TurboJ compiler due to its goal of reducing memory requirements at the expense of execution time. FAJITA does outperform the interpreter, however.

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 [23]. The resulting performance was compared with the same codes executed on Intel's 486 and Pentium processors using both a JIT compiler and an interpreter. The results show that picoJava-I executes the Java codes 15 to 20 times faster than a 486 with an interpreter, and 5 times faster than a Pentium with a JIT

compiler, 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.

The simulation-based performance analysis on the proposed Java processor [27] 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*). Performance results for the Java ILP machine [8] are not yet available.

# 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. These 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, however. Therefore, some translation or interpretation must occur at run time, which is time-consuming. 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. While some of these techniques (e.g. direct Java compilers) provide performance close to compiled C/C++ programs, they lose the portability feature 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 with 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. While these processors will execute Java programs much faster than either interpreters or compiled programs, they are of limited use since we cannot use applications written in other programming languages directly on these processors. Thus, there are a wide variety of techniques available for users to execute Java programs.

The choice of a particular Java execution technique will be guided by the requirements of the application program as well as the performance offered by the technique. However, as we have pointed out earlier, the performance evaluation of these various execution techniques are incomplete due to the lack of a standardized set of Java benchmarks. Also, many of the techniques that have been evaluated were not complete implementations of the Java Virtual Machine. Some implement garbage collection and include exception handling, for instance, while others do 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.

## References


[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass. 1986.

[2] E. Armstrong, *HotSpot: A new breed of Virtual Machine*, JavaWorld, March 1998.

[3] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, Mass., 1996.

[4] A. Bik and D. Gannon, *Automatically Exploiting Implicit Parallelism in Java*, Concurrency: Practice and Experience, Vol. 9, No. 6, June 1997, pp.579-619.

[5] M. Ciernak and W. Li, *Just-in-time optimizations for high-performance Java programs*, Concurrency: Practice and Experience, Vol. 9, No. 11, Nov. 1997, pp. 1063-1073.

[6] M. Cierniak and W. Li, *Optimizing Java Bytecodes*, Concurrency: Practice and Experience, Vol. 9, No.6, June 1997, pp. 427-444.

[7] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers, *Vortex: An Optimizing Compiler for Object-Oriented Languages'*, Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 1996, pp. 93-100.

[8] K. Ebcioglu, E. Altman, and E. Hokenek, *A Java ILP Machine Based on Fast Dynamic Compilation*, International Workshop on Security and Efficiency Aspects of Java, Eilat, Israel, Jan. 9-10, 1997.

[9] D. Flanagan, *Java in a Nutshell*, O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[10] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, 1996.

[11] J. Gosling, *The Feel of Java*, Computer, June 1997, pp. 53-57.

[12] T. R. Halfhill, *How to Soup Up Java*, BYTE magazine, Vol. 23, No. 5, May 1998, pp. 60-74.

[13] C.A. Hsieh, M.T. Conte, J.C. Gyllenhaal, and W.W. Hwu, *Optimizing NET Compilers for Improved Java Performance*, Computer, June 1997, pp. 67-75.

[14] C.A. Hsieh, J.C. Gyllenhaal, and W.W. Hwu, *Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results*, International Symposium on Microarchitecture MICRO 29, 1996, pp. 90-97.

[15] L. Kale, M. Bhandarkar, and T. Wilmarth, *Design and Implementation of Parallel Java with Global Object Space*,Proceedings of Conference on Parallel and Distributed Processing Technology and Applications, Las Vegas, Nevada, 1997.

[16] M. Watheq El-Kharashi and F. Elguibaly, *Java Microprocessors: Computer Architecture Implications*, IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM), Vol. 1, pp. 277-280.

[17] Andreas Krall and Reinhard Grafl, *CACAO - A 64 bit JavaVM Just-in-Time Compiler*, Principles & Practice of Parallel Programming (PPoPP) '97 Java Workshop.

[18] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.

[19] C. Mangione, *Performance tests show Java as fast as C++*, JavaWorld, February 1998.

[20] H. McGhan and M. O'Connor, *PicoJava: A Direct Execution Engine for Java Bytecode*, IEEE Computer, October 1998, pp. 22-30.

[21] J. Meyer and T. Downing, *Java Virtual Machine*, O'Reilly & Associates, Inc., Sebastopol, CA, 1997.

[22] G. Muller, B. Moura, F. Bellard, and C. Consel, *Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code*, Conference on Object-Oriented Technologies and Systems (COOTS) 1997.

[23] J. M. O'Connor and M. Tremblay, *PicoJava-I: The Java Virtual Machine in Hardware*, IEEE Micro, March/April 1997, pp. 45-53.

[24] T.A. Proebsting, et al, *Toba: Java For Applications A Way Ahead of Time (WAT) Compiler*, Conference on Object-Oriented Technologies and Systems (COOTS) 1997.

[25] J. Turley, *MicroJava Pushes Bytecode Performance - Sun's MicroJava 701 Based on New Generation of PicoJava Core*, Microprocessor Report, Vol. 11, No. 15, Nov. 17, 1997.

[26] P. Tyma, *Tuning Java Performance*, Dr. Dobb's Journal of Software Tools for the Professional Programmer, Vol. 21, No. 4, April 1996, pp. 52.

[27] N. Vijaykrishnan, N. Ranganathan, and R. Gadekarla, *Object-Oriented Architectural Support for a Java Processor*, Proceeding of the 12th European Conference on Object-Oriented Programming (ECOOP) 1998, pp. 330-354.

[28] Paul R. Wilson, *Uniprocessor Garbage Collection Techniques*, Proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, Springer-Verlag, September 17-19, 1992, pp. 1-42.

[29] M. J. Wolfe, *High Performance Compilers for Parallel Computers*, Addison-Wesley, Redwood City, CA, 1996.

## Web

## References


[W1] Sun Microsystems - The Source for Java Technology, http://java.sun.com.

[W2] Microsoft compiler for Java, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm/jvc.htm.

[W3] The Java Interpreter, http://java.sun.com/products/jdk/1.0.2/tools/solaris/java.html.

[W4] Microsoft SDK Tools, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm.

[W5] Just-In-Time Compilation and the Microsoft VM for Java, http://premium.microsoft.com/msdn/library/sdkdoc/java/htm/Jit_Structure.htm.

[W6] Netscape Navigator Java Support, http://live.netscape.com/comprod/prod-ucts/navigator/version3.0/java.

[W7] Microsoft Internet Explorer 4.0 Features, http://www.microsoft.com/ie/ie40/features.

[W8] Symantec - Just In Time Compiler Performance Analysis, http://www.symantec.com/jit/jit_pa.html.

[W9] Silicon Graphics - Boosting Java Perfromance, http://www.sgi.com/Products/DevMagic/products/javaperf.html.

[W10] T. Wilkinson, Kaffe v0.10.0 - A Free Virtual Machine to Run Java Code, March 1997. Available at http://www.kaffe.org/.

[W11] SunSoft, http:// www.javaworld.com/javaworld/jw-09-1996/jw-09-workshop.html.

[W12] Digital Java Power, http://www.digital.com/java/.

[W13] JDK for OS/2 (IBM), http://www.cc.usart.ru/Pub/OS2JAVA/catalog/java.htm.

[W14] JDK for AIX (IBM), http://www.cc.usart.ru/Pub/OS2JAVA/catalog/aixjdk.htm.

[W15] Guava JIT Java Runtime, http://www.ph-erfurt.de/information/java/ad/soft/Guava/.

[W16] FAJITA Java to Native Compiler, http://www.gr.opengroup.org/compiler/fajita.htm.

[W17] Supercede - Real World Java Solutions, http://www.supercede.com/prodserv/scfjava.html.

[W18] IBM High Performance Compiler for Java: An Optimizing Native Code Compiler for Java Applications, http://www.alphaWorks.ibm.com/ graphics.nsf/system/graphics/HPCJ/$file/highpcj.html

[W19] IBM Research, http://www.research.ibm.com/topics/innovate/java/.

[W20] j2c/CafeBabe Java .class to C translator, http://www.webcity.co.jp/info/andoh/java/j2c.html.

[W21] TurboJ Java to Native Compiler, http://www.gr.opengroup.org/compiler/turboj.htm.

[W22] TurboJ High Performance Java Compiler, http://www.camb.opengroup.org/openitsol/turboj/technical/benchmarks.htm.

[W23] UW Cecil/Vortex Project, http://www.cs.washington.edu/research/projects/cecil.

[W24] Java Compiler Technology, http://www.gr.opengroup.org/compiler/index.htm.

[W25] The Open Group Research Institute, http://www.camb.opengroup.org/RI/.

[W26] JOLT Project Home Page, http://www.redhat.com/linux-info/jolt/index.html.

[W27] Welcome to Harissa, http://www.irisa.fr/compose/harissa/.

[W28] DashO Pro, http://www.preemptive.com/DashO/.

[W29] Sun Microsystems. MicroJava-701 Processor, http://www.sun.com/microelectronics/microJava-701.

[W30] Patriot Scientific Corporation, Java on Patriot's PSC1000 Microprocessor, http://www.ptsc.com/PSC1000/java_psc1000.html.

[W31] Rockwell's Java Processor JEM1, http://www.rockwell.com/News/PressRel/PR970923.html.

[W32] Java Microbenchmarks, http://www.cs.cmu.edu/~ jch/java/benchmarks.html.

[W33] UCSD Benchmarks for Java, http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.

[W34] The Standard Performance Evaluation Corporation (SPEC), http://www.spec.org/.

[W35] SPEC CPU92 Benchmarks, http://open.specbench.org/osg/cpu92/.

[W36] SPEC CPU95 Benchmarks, http://open.specbench.org/osg/cpu95/.

[W37] Linpack Benchmark - Java Version, http://www.netlib.org/benchmark/linpackjava/.

[W38] CaffeineMark Java Benchmark, http://www.webfayre.com/cm.html.

[W39] PC Magazine Test Center: JMark 1.01, http://www8.zdnet.com/pcmag/pclabs/bench/benchjm.htm.

[W40] Welcome to JMark 2.0, http://www.zdnet.com/zdbop/jmark/jmark.html.

[W41] SYSmark J, http://www.bapco.com/sysmarkj/SYSmarkJ.html.

[W42] VolanoMark, http://www.volano.com/mark.html.

[W43] TurboJ Benchmark's Results, http://www.camb.opengroup.org/openitsol/turboj/technical/benchmarks.htm.

[W44] Dhrystone Benchmark, http://www.netlib.org/benchmark/dhry-c.

[W45] Dhrystone Benchmark in Java, http://www.c-creators.co.jp/okayan/DhrystoneApplet/.

[W46] Jell: Parser Generator, http://www.sbktech.org/jell.html.

[W47] Jax: Scanner Generator, http://www.sbktech.org/jax.html.

[W48] Jas: Bytecode Assembler, http://www.sbktech.org/jas.html.

[W49] EspressoGrinder, http://wwwipd.ira.uka.de/~ espresso/.

[W50] Java Benchmarking, http://www-csag.cs.uiuc.edu/individual/a-coday/personal/projects/java/java-bench.html.

[W51] Java Benchmark References, http://www.ee.umd.edu/femtojava/presentations/97/benchmark.html.

[W52] JLex: A Lexical Analyzer Generator for Java, http://www.cs.princeton.edu/~ appel/modern /java/JLex.

[W53] CUP Parser Generator for Java, http://www.cs.princeton.edu/~ appel/modern/java/CUP.

[W54] JHLZip - another zippy utility, http://www.easynet.it/~ jhl/apps/zip/zip.html.

[W55] JHLUnzip - a zippy utility, http://www.easynet.it/~ jhl/apps/zip/unzip.html.