The Design and Performance of MedJava

Experience Developing Performance-Sensitive Distributed Applications with Java

Prashant Jain, Seth Widoff, and Douglas C. Schmidt§

Department of Computer Science, Washington University, St. Louis, MO, 63130

Abstract. The Java programming language has gained substantial popularity in the past two years. Java's networking features, along with the growing number of Web browsers that execute Java applets, facilitate Internet programming. Despite the popularity of Java, however, there are many concerns about its efficiency. In particular, networking and computation performance are key concerns when considering the use of Java to develop performance-sensitive distributed applications.

This paper makes three contributions to the study of Java for performance-sensitive distributed applications. First, we describe an architecture using Java and the Web to develop MedJava, which is a distributed electronic medical imaging system with stringent networking and computation requirements. Second, we present benchmarks of MedJava image processing and compare the results to the performance of `xv`, which is an equivalent image processing application written in C. Finally, we present performance benchmarks using Java as a transport interface to exchange large medical images over high-speed ATM networks.

For computationally-intensive algorithms like image filtering, Java code that is optimized both manually and with JIT compilers can sometimes compensate for the lack of compile-time optimizations and yield performance commensurate with equivalent compiled C code. With rigorous compile-time optimizations, however, C compilers still generally generate more efficient code. The advent of highly optimizing Java compilers should make it feasible to use Java for performance-sensitive distributed applications where C and C++ are used currently.

§ To whom correspondence should be addressed.

1. Introduction

This paper examines the feasibility of using Java to develop large-scale distributed applications with demanding performance requirements for networking and processing speed. To bound the scope of our investigation, we focus on the design and performance of *MedJava*, which is a distributed electronic medical imaging system (EMIS). MedJava uses Java and Web tools to perform image processing on medical images distributed throughout an EMIS.

Image processing is a set of computational techniques for enhancing and analyzing images. Image processing techniques apply algorithms, called *image filters*, to manipulate images. For example, radiologists may need to sharpen an image to properly diagnose a tumor. Similarly, to identify a kidney stone, a radiologists may need to zoom into an image while maintaining high resolution. Thus, an EMIS must provide powerful image processing capabilities, as well as efficient distributed image retrieval and storage mechanisms.

To evaluate Java's image processing performance, we conducted extensive benchmarking of MedJava and compared the results to the performance of `xv`, an equivalent image processing application written in C. To evaluate the performance of Java as a transport interface for exchanging large images over high-speed networks, we performed a series of network benchmarking tests over a 155 Mbps ATM switch and compared the results to the performance of C/C++ as a transport interface.

Our empirical measurements reveal that an imaging system implemented in C/C++ always out-performs an imaging system implemented using interpreted Java by 30 to 100 times. However, the performance of Java code using a "just-in-time" (JIT) compiler is ~1.5 to 5 times slower than the performance of compiled C/C++ code. Likewise, using Java as the transport interface performs 2% to 50% slower than using C/C++ as the transport interface. However, for sender buffer size close to the network MTU size, the performance of using Java as the transport interface was only 9% slower than the performance of using C/C++ as the transport interface. Therefore, we conclude that it is becoming feasible to use Java to develop large-scale distributed EMISs. Java is particularly relevant for wide-area environments, such as teleradiology, where conventional EMIS capabilities are too costly or unwieldy with existing development tools.

The remainder of this paper is organized as follows: Section 2 motivates and describes the object-oriented (OO) design and features of MedJava; Section 3 compares the performance of MedJava with an equivalent image processing application written in C and compares the performance of a Java transport interface with the performance of a C/C++ transport interface; Section 4 describes related work; and Section 5 presents concluding remarks.

2. Design of the MedJava Framework

2.1. Motivation

Medical imaging plays a key role in the development of a regulatory review process for radiologists and physicians [1]. The demand for electronic medical imaging systems (EMISs) that allow visualization and processing of medical images has increased significantly [2]. This increase is due to the advent of modalities, such as angiography, magnetic resonance imaging (MRI), computed tomography (CT), nuclear medicine, and ultrasound, that acquire data digitally and the ability to digitize medical images from film.

The growing demand for EMISs has also been coupled with a need to access medical images and other diagnostic information remotely across networks [3]. Connecting radiologists electronically with patients increases the availability of health care. In addition, it can facilitate the delivery of remote diagnostics and remote surgery [4].

As a result of these forces, there is also increasing demand for *distributed* EMISs. These systems supply health care providers with the capability to access medical images and related clinical studies across a network in order to analyze and diagnose patient records and exams. The need for distributed EMISs is also driven by economic factors. As independent health hospitals consolidate into integrated health care delivery systems [2], they will require distributed computer systems to unify their multiple and distinct image repositories.

Figure 1 shows the network topology of a distributed EMIS. In this environment, medical images are captured by modalities and transferred to appropriate Image Stores. Radiologists and physicians can then download these images to diagnostic workstations for viewing, image processing, and diagnosis. High-speed networks, such as ATM or Fast Ethernet, allow the transfer of images efficiently, reliably, and economically.

2.2. Problem: Resolving Distributed EMIS Development Forces

A distributed EMIS must meet the following requirements:

- *Usable:* An EMIS must be usable to make it as convenient to practice radiology as conventional film-based technology.

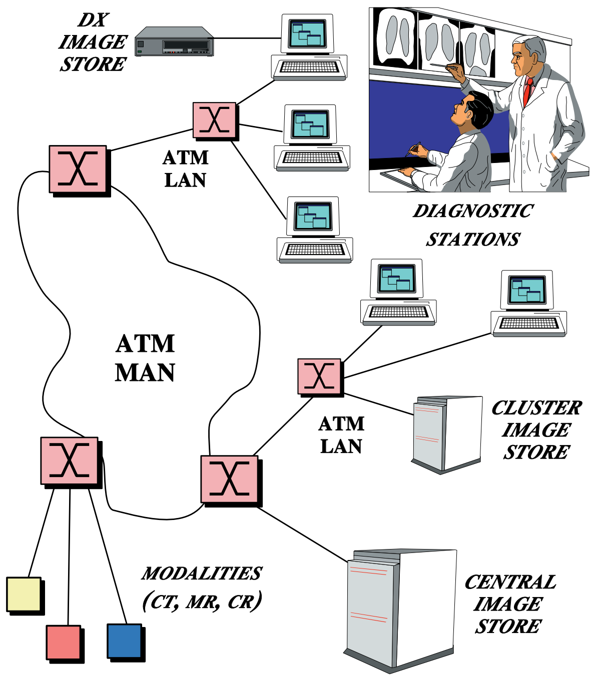

Figure 1. Topology of a Distributed EMIS

- *Efficient:* An EMIS must be efficient to process and deliver medical images rapidly to radiologists.

- *Scalable:* An EMIS must be scalable to support the growing demands of large-scale integrated health care delivery systems [2].

- *Flexible:* An EMIS must be flexible to transfer different types of images and to dynamically reconfigure image processing features to cope with changing requirements.

- *Reliable:* An EMIS must be reliable to ensure that medical images are delivered correctly and are available when requested by users.

- *Secure:* An EMIS must be secure to ensure that confidential patient information is not compromised.

- *Cost-effective:* An EMIS must be cost-effective to minimize the overhead of accessing patient data across networks.

Developing a distributed EMIS that meets all of these requirements is challenging, particularly since certain features conflict with other features. For example, it is hard to develop an EMIS that is efficient, scalable, and cost-effective. This is because efficiency often requires high-performance computers and high-speed networks, thereby raising costs as the number of system users increases.

2.3. Solution: Java and the Web

Over the past two years, the Java programming language has sparked considerable interest among software developers. Its popularity stems from its flexibility, portability, and relative simplicity compared with other object-oriented programming languages [5].

The strong interest in the Java language has coincided with the ubiquity of inexpensive Web browsers. This has brought the Web technology to the desktop of many computer users, including radiologists and physicians.

A feature supported by Java that is particularly relevant to distributed EMISs is the *applet*. An applet is a Java class that can be downloaded from a Web server and run in a context application such as a Web browser or an applet viewer. The ability to download Java classes across a network can simplify the development and configuration of efficient and reliable distributed applications [6].

Once downloaded from a Web server, applets run as applications within the local machine's Java run-time environment, which is typically a Web browser. In theory, therefore, applets can be very efficient since they harness the power of the local machine on which they run, rather than requiring high latency RPC calls to remote servers [7].

The MedJava distributed EMIS was developed as a Java applet. Therefore, it exploits the functionality of front-ends offered by Web browsers. An increasing number of browsers (such as Internet Explorer and Netscape Navigator and Communicator) are Java-enabled and provide a run-time environment for Java applets. A Java-enabled browser provides a Java Virtual Machine (JVM), which is used to execute Java applets. MedJava leverages the convenience of Java to manipulate images and provides image processing capabilities to radiologists and physicians connected via the Web.

In our experience, developing a distributed EMIS in Java is relatively cost effective since Java is fairly simple to learn and use. In addition, Java provides standard packages that support GUI development, networking, and image processing. For example, the package `java.awt.image` contains reusable classes for managing and manipulating image data, including color models, cropping, color filtering, setting pixel values, and grabbing bitmaps

[8].

Since Java is written to a virtual machine, an EMIS developer need only compile the Java source code to Java bytecode. The EMIS applet will execute on any platform that has a Java Virtual Machine implementation. Many Java bytecode compilers and interpreters are available on a variety of platforms. In principle, therefore, switching to new platforms or upgraded hardware on the same platform should not require changes to the software or even recompilation of the Java source. Consequently, an EMIS can be constructed on a network of heterogeneous machines and platforms with a single set of Java class files.

2.4. Caveat: Meeting EMIS Performance Requirements

Despite the software engineering benefits of developing a distributed EMIS in Java, there are serious concerns with its performance relative to languages like C and C++. Performance is a key requirement in a distributed EMIS since timely diagnosis of patient exams by radiologists can be life-critical. For instance, in an emergency room (ER), patient exams and medical images must be delivered rapidly to radiologists and ER physicians. In addition, an EMIS must allow radiologists to process and analyze medical images efficiently to make appropriate diagnoses.

Meeting the performance demands of a large-scale distributed EMIS requires the following support from the JVM. First, its image processing must be precise and efficient. Second, its networking mechanisms must download and upload large medical images rapidly. Assuming that efficient image processing algorithms are used, the performance of a Java applet depends largely on the efficiency of the hardware and the JVM implementation on which the applet is run.

The need for efficiency motivates the development of high-speed JIT compilers that translate Java bytecode into native code for the local machine the browser runs on. JIT compilers are “just-in-time” since they compile Java bytecode into native code on a per-method basis immediately before calling the methods. Several browsers, such as Netscape and Internet Explorer, provide JIT compilers as part of their JVM.

Although Java JIT compilers avoid the penalty of interpretation, previous studies [9] show that the cost of compilation can significantly interrupt the flow of execution. This performance degradation can cause Java code to run significantly slower than compiled C/C++ code. Section 3 quantifies the overhead of Java and C/C++ empirically.

2.5. Key Features of MedJava

MedJava has been developed as a Java applet. Therefore, it can run on any Java-enabled browser that supports the standard AWT windowing toolkit. MedJava allows users to download medical images across the network. Once an image has been downloaded, it can be processed by applying one or more image filters, which are based on algorithms in the C source code from `xv`. For example, a medical image can be sharpened by applying the Sharpen Filter. Sharpening a medical image enhances the details of the image, which is useful for radiologists who diagnose internal ailments.

Although MedJava is targeted for distributed EMIS requirements, it is a general-purpose imaging tool that can process both medical and non-medical images. Therefore, in addition to providing medical filters like sharpening or unsharp masking, MedJava provides other non-medical image processing filters such as an Emboss filter, Oil Paint filter, and Edge Detect filter. These filters are useful for processing non-medical images. For example, edge detection serves as an important initial step in many computer vision processes because edges contain the bulk of the information within an image [10]. Once the edges of an image are detected, additional operations such as pseudo-coloring can be applied to the image.

Image filters can be dynamically configured and re-configured into MedJava via the *Service Configurator* pattern [6]. This makes it convenient to enhance filter implementation or install new filters without restarting the MedJava applet. For example, a radiologist may find a sharpen filter that uses the unsharp mask algorithm to be more efficient than a sharpen filter that simply applies a convolution matrix to all the pixels. Doing this substitution in MedJava is straightforward and can be done without reloading the entire applet.

Once an image has been processed by applying the filter(s), it can be uploaded to the server where the applet was downloaded. HTTP server implementations, such as JAWS [11, 12] and Jigsaw, support file uploading and can be used by MedJava to upload images. In addition, the MedJava applet provides a hierarchical browser that allows users to traverse directories of images on remote servers. This makes it straightforward to find and select images across the network, making MedJava quite usable, as well as easy to learn.

To facilitate performance measurements, the MedJava applet can be configured to run in benchmark mode. When the applet runs in benchmark mode, it computes the time (in milliseconds) required to apply filters on downloaded images. The timer starts at the beginning of each image processing algorithm and stops immediately after the algorithm terminates.

2.6. The OO Design of MedJava

Figure 2 shows the architecture of the MedJava framework developed at Washington University to meet distributed EMIS requirements. The two primary components in the

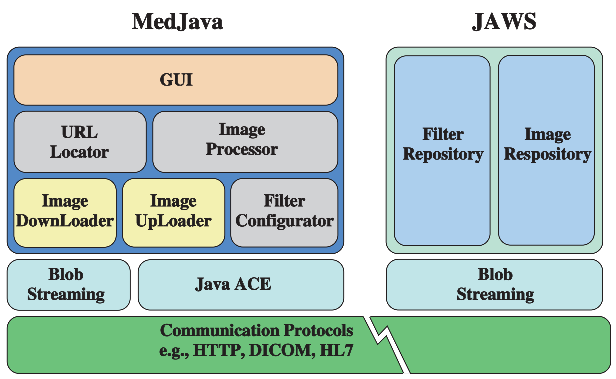

Figure 2. MedJava Framework

architecture include the MedJava client applet and JAWS, which is a high-performance HTTP server also developed at Washington University [12, 11]. The MedJava applet was implemented with components from Java ACE [13], the Blob Streaming framework [14], and standard Java packages such as `java.awt` and `java.awt.image`. Each of these components is outlined below.

2.6.1. MedJava Applet The MedJava client applet contains the following components shown in Figure 2:

Graphical User Interface: which provides a front-end to the image processing tool. Figure 3 illustrates the graphical user interface (GUI) used to display a podiatry image. The MedJava GUI allows users to download images, apply image processing filters on them, and upload the images to a server.

URL Locator: which locates a URL that can reference an image or a directory. If the URL points to a directory, the contents of the directory are retrieved so users can browse them to obtain a list of images and subdirectories in that directory. The URL Locator is used by the Image Downloader and Image Uploader to download and upload images, respectively.

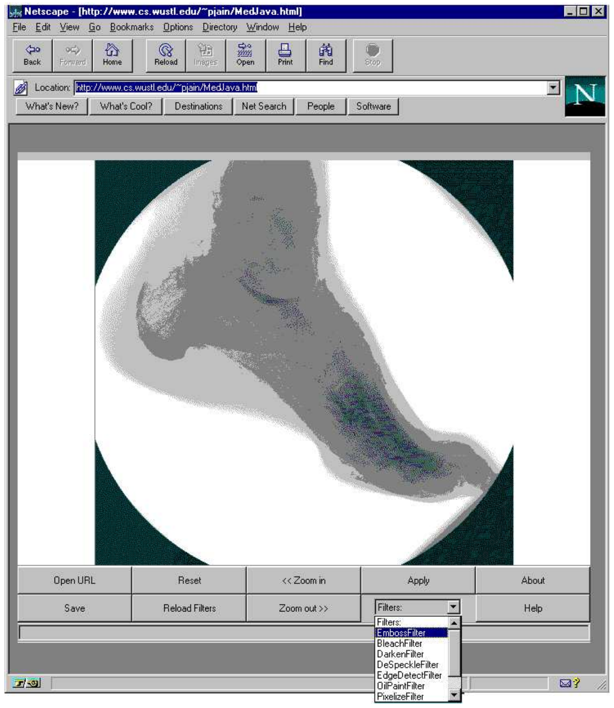

Figure 3. Processing a Medical Image in MedJava

Image Downloader: which downloads an image located by the URL Locator and displays the image in the applet. The Image Downloader ensures that all pixels of the image are retrieved and displayed properly.

Image Processor: which processes the currently displayed image using the image filter selected by the user. Processing an image manipulates the pixel values of the image to create and display a new image.

Image Uploader: which uploads the currently displayed image to the server from where the applet was downloaded from.|| The Image Uploader generates a GIF-format for the currently displayed image and writes the data to the server. This allows the user to save processed images persistently at the server.

Filter Configurator: which downloads image filters from the Server and configures them in the applet. The Filter Configurator uses the Service Configurator pattern [6] to dynamically configure the image filters.

|| Due to applet security restrictions, images can only be uploaded to the server where the applet was downloaded from. In addition, the Web server must support file uploading by implementing the HTTP `PUT` method.

2.6.2. JAWS JAWS is a high-performance, multi-threaded, HTTP Web Server [11]. For the purposes of MedJava, JAWS stores the MedJava client applet, the image filter repository, and the images. The MedJava client applet uses the image filter repository to download specific image filters. Each image filter is a Java class that can be downloaded by MedJava. This design allows MedJava applets to be dynamically configured with image filters, thereby making image filter configuration highly flexible.

In addition, JAWS supports file uploading by implementing the HTTP `PUT` method. This allows the MedJava client applet to save processed images persistently at the server. JAWS implements other HTTP features (such as CGI bin and persistent connections) that are useful for developing Web-based systems.

Figure 4 illustrates the interaction of MedJava and JAWS. The *MedJava client applet*

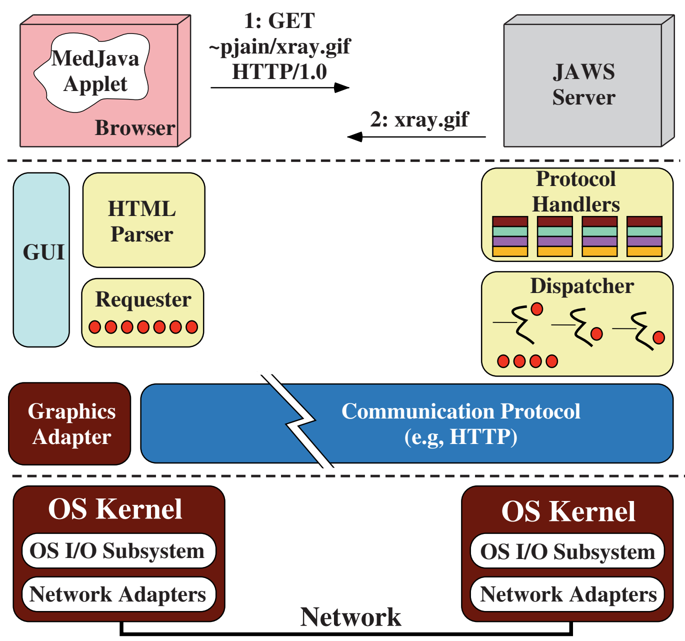

Figure 4. Architecture of MedJava and JAWS

is downloaded into a Web browser from the *JAWS server*. Through *GUI* interactions, a radiologist instructs the MedJava client applet to retrieve images from JAWS (or other servers across the network). The *requester* is the active component of the browser running the MedJava applet that communicates over the *network*. It issues a request for the image to JAWS with the appropriate syntax of the *transfer protocol* (which is HTTP in this case). Incoming requests to the JAWS are received by the *dispatcher*, which is the request demultiplexing engine of the server. It is responsible for creating new threads. Each request is processed by a *handler*, which goes through a *lifecycle* of parsing the request, logging the request, fetching image status information, updating the cache, sending the image, and cleaning up after the request is done. When the response returns to the client with the requested image, it is parsed by an *HTML parser* so that the image may be rendered. At this stage, the *requester* may issue other requests on behalf of the client, *e.g.*, to maintain a client-side cache.

2.6.3. Blob Streaming Figure 5 illustrates the Blob Streaming framework. The framework

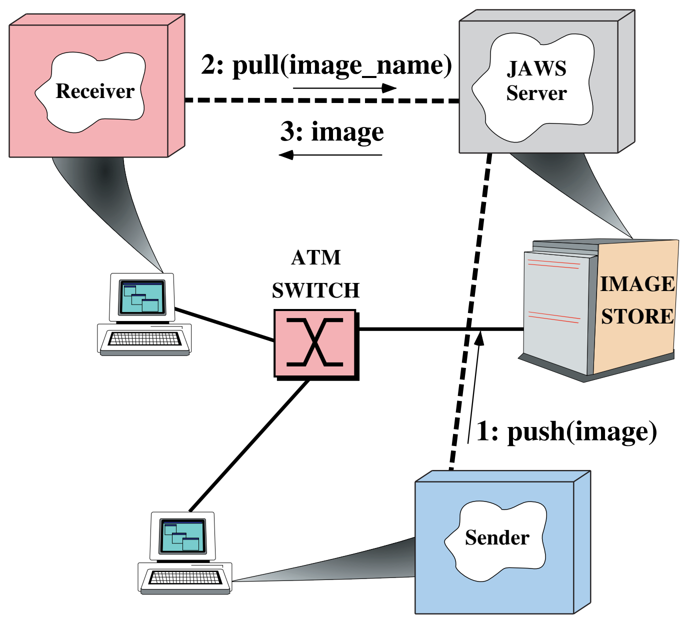

Figure 5. Blob Streaming Framework

provides a uniform interface that allows EMIS application developers to transfer data across a network flexibly and efficiently. Blob Streaming uses the HTTP protocol for the data transfer.¶ Therefore, it can be used to communicate with high-performance Web servers (such as JAWS) to download images across the network. In addition, it can be used to communicate with Web servers that implement the HTTP `PUT` method to upload images from the browser to the server.

Although Blob Streaming supports both image downloading and image uploading across the network, its use within a Java applet is restricted due to applet security mechanisms. To prevent security breaches, Java imposes certain restrictions on applets. For example, a Java applet can not write to the local file system of the local machine it is running on. Similarly, a Java applet can generally download files only from the server where the applet

¶ Although the current Blob Streaming protocol is HTTP, other medical-specific communication protocols (such as DICOM and HL7) can also be supported.

was downloaded. Likewise, a Java applet can only upload files to the server where the applet was downloaded.

Java applets provide an exception to these security restrictions, however. In particular, the Java `Applet` class provides a method that allows an applet to download images from any server reachable via a URL. Since the method is defined in the Java `Applet` class, it allows Java to ensure there are no security violations. MedJava uses this `Applet` method to download images across the network. Therefore, images to be processed can reside in a file system managed by the HTTP Server from where the MedJava client applet was downloaded or can reside on some other server in the network. However, Blob Streaming can only be used to upload images to the server where the MedJava applet was downloaded.

2.6.4. Java ACE Java ACE [5] is a port of the C++ version of the ADAPTIVE Communication Environment (ACE) [15]. ACE is an OO network programming toolkit that provides reusable components for building distributed applications. Containing ∼125,000 lines of code, the C++ version of ACE provides a rich set of reusable C++ wrappers and framework components that perform common communication software tasks portably across a range of OS platforms.

The Java version of ACE Contains ∼10,000 lines of code, which is over 90% smaller than the C++ version. The reduction in size occurs largely because the JVM provides most of the OS-level wrappers necessary in C++ ACE. Despite the reduced size, Java ACE provides most of the functionality of the C++ version of ACE, such as event handler dispatching, dynamic (re)configuration of distributed services, and support for concurrent execution and synchronization. Java ACE implements several key design patterns for concurrent network programming, such as Acceptor and Connector [16] and Active Object [17]. This makes it easier to developing networking applications using Java ACE compared to programming directly with the lower-level Java APIs.

Figure 6 illustrates the architecture and key components in Java ACE. MedJava uses several components in Java ACE. For example, Java ACE provides an implementation of the Service Configurator pattern [6]. MedJava uses this pattern to dynamically configure and reconfigure image filters. Likewise, MedJava uses Java ACE profile timers to compute performance in benchmark mode.

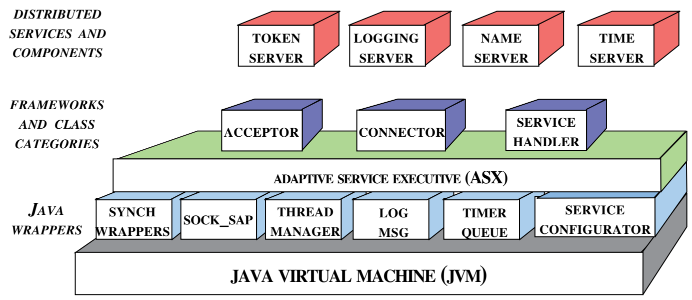

Figure 6. The Java ACE Framework

3. Performance Benchmarks

This section presents the results of performance benchmarks conducted with the MedJava image processing system. We performed the following two sets of benchmarks:

1. Image processing performance: We measured the performance of MedJava to determine the overhead of using Java for image processing. We compared the performance of our MedJava applet with the performance of `xv`. `Xv` is a widely-used image processing application written in C. The MedJava image processing applets are based on the `xv` algorithms.

2. High-speed networking performance: We measured the performance of using Java sockets over a high-speed ATM network to determine the overhead of using Java for transporting data. We compared the network performance results of Java to the results of similar tests using C/C++.

Below, we describe our benchmarking testbed environment, the benchmarks we performed, and the results we obtained.

3.1. MedJava Image Processing Benchmarks

We benchmarked MedJava to compare the performance of Java with `xv`, which is a widely-used image processing application written in C. `Xv` contains a broad range of image filters such as Blur, Sharpen, and Emboss. By applying a filter to an image in `xv`, and then applying an equivalent filter algorithm written in Java to the same image, we compared the performance of Java and C directly. In addition, we benchmarked the performance of different Web browsers running the MedJava applet.

3.1.1. Benchmarking Testbed Environment

Hardware Configuration: To study the performance of MedJava, we constructed a hardware and software testbed consisting of a Web server and two clients connected by Ethernet, as shown in Figure 7. The clients in our experiment were Micron Millenia PRO2 plus

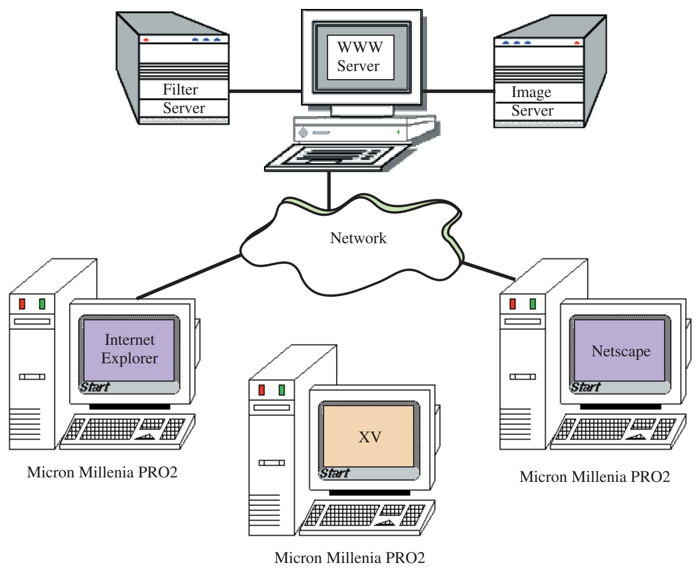

Figure 7. Web Browser Testbed Environment

workstations. Each PRO2 has 128 MB of RAM and is equipped with dual 180 Mhz PentiumPro processors.

JVM Software Configuration: We ran MedJava in two different Web browsers to determine how efficiently these browsers execute Java code. The browsers chosen for our tests were Internet Explorer 4.0 release 2 on Windows NT and Netscape 4.0 on NT. Internet Explorer 4.0 on NT and Netscape 4.0 on NT include Java JIT compilers, written by Microsoft and Symantec, respectively.

As shown in Section 3.1.4, JIT compilers have a substantial impact on performance. To compare the performance of the `xv` algorithms with their Java counterparts, we extracted the GIF loading and processing elements from the freely distributed `xv` source, removed all remnants of the X-Windows GUI, instrumented the algorithms with timer mechanisms in locations equivalent to the Java algorithms. We compiled this subset of `xv` using Microsoft Visual C++ version 5.0, with full optimization enabled.

Image filters can require $O(n^2)$ time to execute, if the filter dimensions approach the image dimensions. For large images, this processing can dominate the loading and display times. Therefore, the running time of the algorithms is an appropriate measure of the overall performance of an image processing application.

Image processing configuration: The standard Java image processing framework uses a "Pipes and Filters" pattern architecture [18]. Downstream sits an `java.image.ImageConsumer` that has registered with an upstream `java.image.ImageProducer` for pixel delivery. The `ImageProducer` invokes the `setPixels` method on the `ImageConsumer`, delivering portions of the image array until it completes by invoking the `ImageComplete` method.

The Pipes and Filters pattern architecture allows the `ImageConsumer` subclass to process the image as it receives the pieces or when the image source arrives in its entirety. An `ImageFilter` is a subclass of `ImageConsumer` situated between the producer and consumer who intercepts the flow of pixels, altering them in some way before it passes the image to the subsequent `ImageConsumer`. All `ImageFilters` in this experiment override the `ImageComplete` method and iterate over each pixel. The computational complexity for each filter depends on how much work the filter does during each iteration.

We selected the following seven filters, which exhibit different computational complexities. These filters are available in both `xv` and MedJava, and are ranked according to their usefulness in the domain of medical image processing.

1. Sharpen Filter: which computes for each pixel the mean of the "values" of the 3×3 matrix surrounding the pixel. In the Hue-Saturation-Value color model, the "value" is the maximum of the normalized red, green, and blue values of the pixel; conceptually, the brightness of that pixel. The new value for the pixel is: $\frac{\text{value} - p * (\text{mean value})}{1-p}$, where p is a value between 0 and 1. The filter exaggerates the contrast between a pixel's brightness and the average brightness of the surrounding pixels.

2. Despeckle Filter: which replaces each pixel with the median color in a 3×3 matrix surrounding the pixel. Used for noise reduction, the algorithm gathers the colors in the square matrix, sorts them using an inlined Shell sort, and chooses the median element.

3. Edge Detect Filter: which runs a merging of a pair of convolutions, one that detects horizontal edges, and one that detects vertical edges. The convolution is done separately for each plane (red, green, blue) of the image, so where there are edges in the red plane, for example, the resultant image will highlight the red edges.

4. Emboss filter: which applies a 3×3 convolution matrix to the image, a variation of an edge detection algorithm. Most of the image is left as a medium gray, but leading and trailing edges are turned lighter and darker gray, respectively.

5. Oil Paint Filter: which computes a histogram of a 3×3 matrix surrounding the pixel and chooses the most frequently occuring color in the histogram to supplant the old pixel value. The result is a localized smearing effect.

6. Pixelize Filter: which replaces each pixel in each 4×4 squares in the image with the average color in the square matrix.

7. Spread Filter: which replaces each pixel by a random one within a 3×3 matrix surrounding the pixel.

8. Blur Filter: which runs a convolution over each plane (red, green, blue) of the image, using a 3×3 convolution mask consisting of all 1's. Effectively, the filter replaces each pixel in the image with the mean of the pixel values in the 3×3 region surrounding the target pixel.

Figure 8 illustrates the original image and processed images that result from applying four of the filters described above. Although some of these filters are not necessarily useful in the medical domain, they follow the same pattern of spatial image processing: the traversal or convolution of a fixed size or variable size matrix over pixels surrounding each pixel in the image array. In principle, therefore, the performance of this set of filters reflect the performance of other more relevant filters of comparable complexity.

3.1.2. Performance Metrics Using the testbed described in Section 3.1.1, we measured the performance of MedJava and compared it with a Windows NT port of `xv`. These tests sent an 8-bit image through each of our eight `xv` filters 10 times, keeping the average of the trials. To test how the performance changed with the image size, we repeated the experiment with the image scaled to different sizes. Both `xv` and Java convert 8-bit images, either greyscale or color, into 24-bit RGB color images prior to filtering. Moreover, all eight algorithms

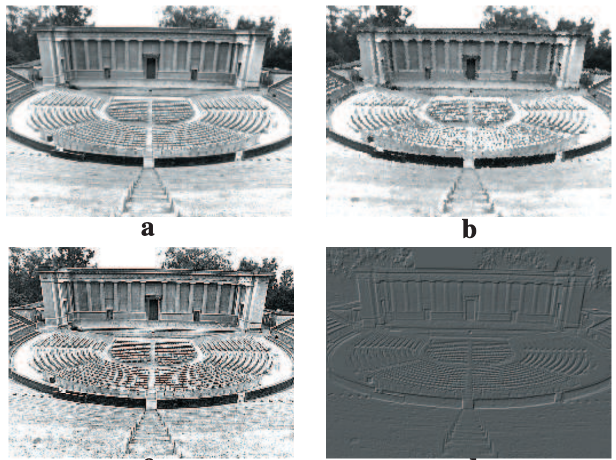

Figure 8. (a) Original Image; (b) Oil-painted Image; (c) Sharpened Image; (d) Embossed Image

are functions solely of image dimension and not pixel value. Thus, there is no processing performance difference between greyscale and color images in either environment.

We expected *a priori* that the C code would outperform the Java filters due to the extensive optimizations performed by the Microsoft Visual C/C++ compiler.+ Therefore, we coded the MedJava image filters using the source level optimization techniques described in Section 3.1.3 to elicit maximum performance from them.

However, contrary to our expectations, the hand optimized Java algorithms performed nearly as well as their C counterparts. Therefore, we also optimized the C algorithms by hand. This rendered the two sets of algorithms nearly indistinguishable in appearance, but not indistinguishable in performance. For MedJava, we ran three trials, one on Internet Explorer 4.0 release 2 on NT (IE 4), one on Netscape Navigator 4.0 on NT (NS 4), and one on Internet Explorer 4.0 release 2 with just-in-time compilation disabled (IE 4 JIT off).

3.1.3. Source Level Optimizations The Java run-time system, including the garbage collector and the Abstract Window Toolkit (AWT), was written using C. Therefore, they cannot be optimized by the Java bytecode interpreter or compiler. As a result, any attempt to improve the performance of Java in medical imaging systems must improve the performance of code spent outside these areas, *i.e.*, in the image filters themselves.

JIT compilers affect the greatest speed up in computationally intensive tasks that do not

+ "Full optimization" on MVC++ includes: inline function expansion, subexpression elimination, automatic register allocation, loop optimization, inlining of common library functions, and machine code optimization.

call the AWT or run-time system, as shown by the benchmarks in Table 1. These benchmarks test the performance of common image filter operations in the two browsers used in the experiments. These data were obtained by wrapping a test harness around a loop that iterates ten times, subtracts the loop overhead from the result, and divides by the number of iterations. Java's garbage collection routine was called before the sequence to prohibit it from affecting the test results. The results are listed in nanoseconds.

Since the conversion from byte-code to native code is already costly, JIT compilers do not spend a great deal time attempting to further optimize the native code. Therefore, lacking source to a bytecode compiler that optimizes its output, the most a developer of performance-critical applications can do to further accelerate the performance of computationally-intensive tasks is to optimize the source code manually. The image filters in MedJava leveraged the following canonical techniques and insight on how to best optimize computationally intensive source code in Java [19]:

Strength reduction: which replaces costly operations with a faster equivalent. For instance, the Image Filters converted multiplications and divides by factors of two into lefts and rights shifts.

Common subexpression elimination: which removes redundant calculations. Image Filters store the pixel values in a one dimensional array. Thus, for each pixel access this optimization calculates a pixel index once from the column and row values and stores the results of the array access into a temporary variable, rather than continually indexing the same element of the array.

Code motion: which hoists constant calculations out of loops. Thus, although it may be impossible to unroll loops where the number of iterations is a function of the image height and width, the Image Filters reduce the overhead of loops by removing constant calculations computed at each loop termination check.

Local variables: which are efficient to access. The virtual machine stores them in an array in the method frame. Thus, there is no overhead associated with dereferencing an object reference, unlike an instance variable, a class name, or a static data member. The bytecodes `getfield` and `getstatic` must first resolve the class and method names before pushing

Operation	NS 4	IE 4	IE 4 JIT off
Loop overhead	10.21	10.21	902
Quick Int Assignment	5.01	5.01	339
Local Int Assignment	5.01	5.32	440
Static Member Integer	25.24	20.13	851
Member Integer	5.01	10.02	560
Reference Assignment	5.01	10.12	580
Integer Array Access	11.63	5.21	350
Static Instance Method	35.45	34.95	692
Instance Method	40.46	30.24	922
Final Instance Method	30.35	40.17	922
Private Instance Method	35.46	30.24	992
Random.nextInt()	80.92	86.71	450
int++	20.33	7.72	180
int = int + int	5.02	10.11	710
int = int - int	15.12	10.12	690
int = int * int	10.12	10.12	691
int = int / int	75.96	71.35	890
int /= 2	16.43	10.92	931
int >>= 1	17.43	7.21	661
int *= 2	19.63	7.42	780
int <<=1	17.33	7.71	731
int = int & int	15.13	5.01	670
int = int | int	5.01	0.11	670
float = float + float	15.12	10.12	730
float = float - float	15.12	10.12	700
float = float * float	10.02	15.22	720
float = float / float	46.82	46.02	841
Cast double to float	4.91	5.01	570
Cast float to int	67.14	347.3	910
Cast double to int	67.15	13.06	920

Table 1. Times in Nanoseconds for Common Operations in the Testbed Java Environment

the value of the variable onto the operand stack. Also, the `iload` and `istore` instructions allow the JVM to quickly load and store the first four local variables to and from the operand stack [20].

Integer variables, floats, and object references: which are most directly supported by the JVM since the operand stack and local variables are each one word in width, the size of integers, floating points, and references. Smaller types, such as `short` and `byte` are not directly supported in the instruction set. Therefore, each must be converted to an `int` prior to an operation and then subsequently back to the smaller type, accruing the cost of a valid truncation [20].

Manually inlining methods: eliminates the overhead associated with method invocation. Although `static`, `final`, and `private` methods can be resolved at compile time, eliminating method calls entirely, especially simple calls on the `java.lang.Math` package (*e.g.*, `ceil`, `floor`, `min`, and `max`), in critical sections of looping code will further improve performance.

The `final`, `static`, or `private` keywords on a method advises the run-time compiler or interpreter that it may safely inline the method. However, because classes are linked together at run-time, changes made to a `final` method in one class would not be reflected in other already compiled classes that invoke that method, unless they too were recompiled [21]. Naturally, when invoking methods internal to a class, this is not a problem. Moreover, the `-O` option on the Sun `javac` source to bytecode compiler requests that it attempt to inline methods.

As an example of worthwhile manual method inlining, an `ImageFilter` contains a method called `setColorModel`. In this method the `ImageProducer` provides the `ImageFilter` with the `ColorModel` subclass that grabs the color values of each pixel in the image source. `ColorModel` is an abstract class with methods `getRed`, `getGreen`, and `getBlue` to retrieve the specific color value from each pixel. Thus, every time a filter needs a color value, it must incur the overhead of this dynamically resolved method call. However, calling the `getDefaultColorModel` static method on `ColorModel` returns a `ColorModel` subclass, which guarantees that each pixel will be in a known form, where the first 8 bits are the alpha (transparency) value, the next 8 are the red, the next 8 are the green, and the last 8 bits are the blue value. Therefore, rather than using the methods on `ColorModel` to retrieve the color values, the `ImageFilter` can retrieve values simply by shifting and masking the integer value of the pixel, *e.g.*, to obtain the red value of a pixel: (pixel >> 16) & 0xff.

Of course, for C code many of the same optimization techniques apply. We ran a similar set of operation benchmarks in C, using the same test harness technique as we did for the Java benchmarks. The results, shown in Table 2, are the mean of 5 trials, with each measurement exhibiting a standard deviation of no more than 0.5 nanoseconds.

operation	MSVC++ output
Local int access	7.40
Extern int access	3.57
Static int access	7.13
Heap byte access	7.62
Heap int access	7.99
Stack byte array	7.72
Stack int array	7.82
Global byte array	8.60
Global int array	8.17
Static function call	25.40
Extern function call	32.02
int++	7.39
int = int + int	11.88
int = int - int	11.88
int = int * int	21.88
int = int / int	203.27
int *= 2	7.18
int <<=1	3.55
int /= 2	13.47
int >>= 1	7.25
int = int & int	11.86
int = int | int	7.68
float = float + float	16.58
float = float - float	16.51
float = float * float	556.08
float = float / float	611.43
Call to rand()	57.33
cast from float to int	863.11

Table 2. Times in Nanoseconds for Common Operations in C/C++ on the Testbed Platform

With "global optimizations" enabled, the MSVC++ compiler will actively assign variables to registers at its own discretion. With optimizations disabled, it takes no special measures to abide by the `register` keyword. Again, the results are listed in nanoseconds.

Table 2 reveals that the MSVC++ generated code yields comparable performance with the output of the two JIT compilers. Narrowing casts, for example from floating point to integer data is more time consuming in the MSVC++ generated code than the JIT output, however, calls to static, external, and library functions (*e.g.*, `rand`) are less time consuming than their Java method equivalents. Also, floating point multiplication and division, translated into the `fmul` and `fdiv` in MSVC++, lag behind the JIT translation of these operations.

3.1.4. Performance Results and Evaluation Figures 9–16 plot our results for each of the eight filters on each of the three language/compiler permutations.

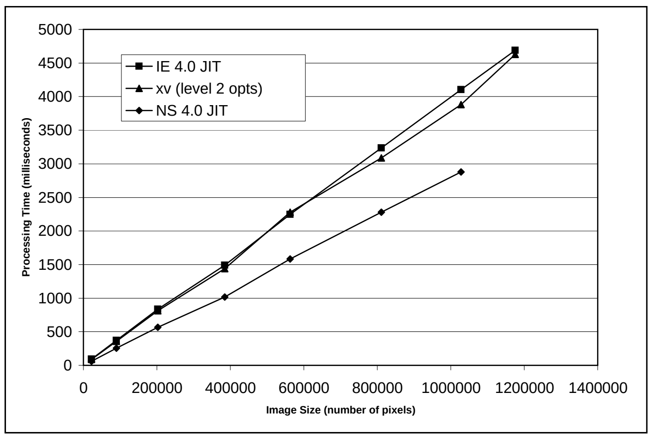

Figure 9. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Sharpen Image Filter to an Image at Various Sizes

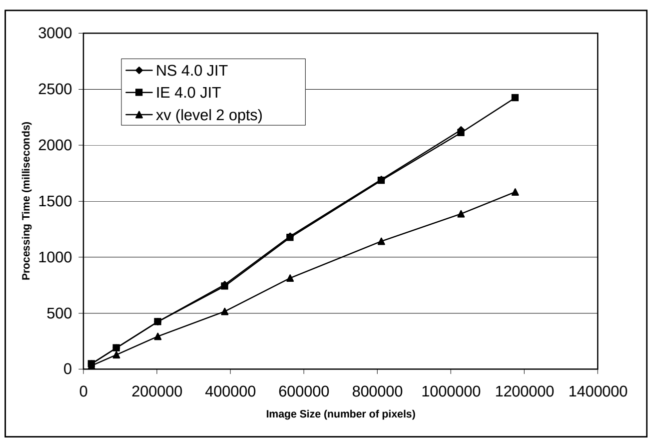

Figure 10. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Edge Detection Image Filter to an Image at Various Sizes

Using insights about the most frequently performed operations in the algorithms, and the tables enumerating the costs of those operations on the three configurations (Tables 1 and 2), we can attempt to explain any observed, counter-intuitive differences in the performance of the algorithms.

In general, the hand-optimized Java algorithms executed in times comparable with their C hand-optimized counterparts. However, the added benefit of the MSVC++ compile-time

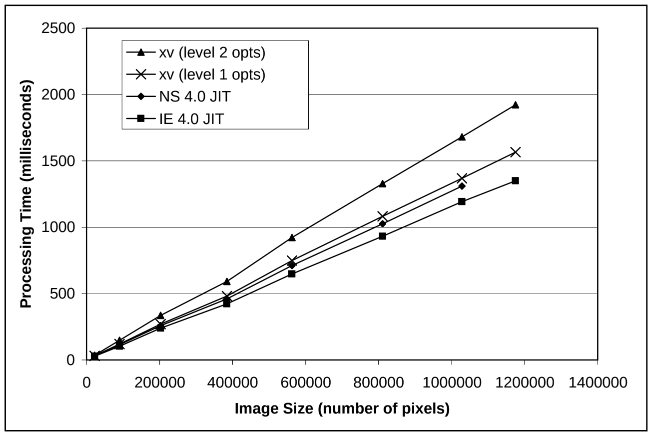

Figure 11. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Blur Image Filter to an Image at Various Sizes

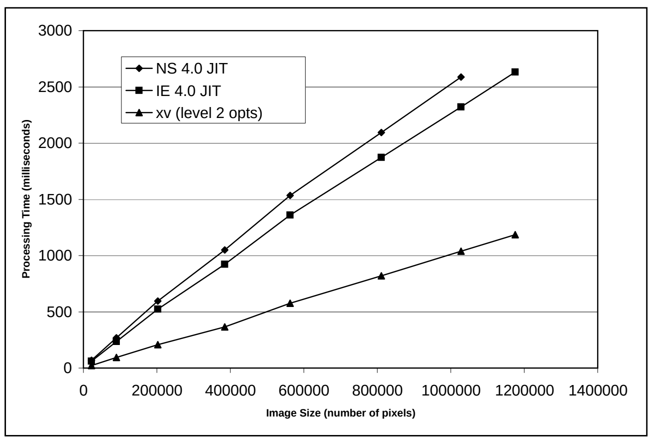

Figure 12. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Despeckle Image Filter to an Image at Various Sizes

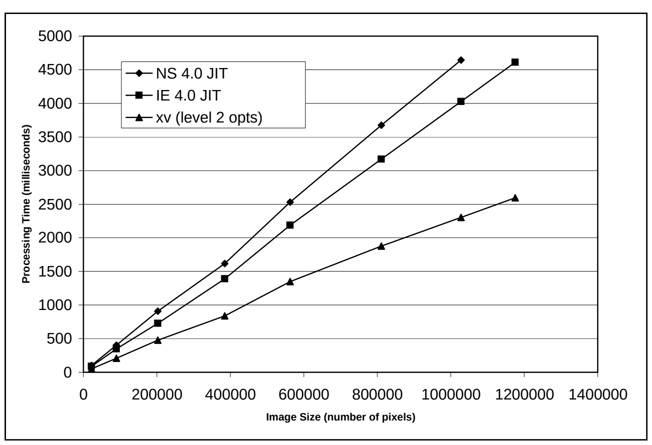

Figure 13. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Oil Paint Image Filter to an Image at Various Sizes

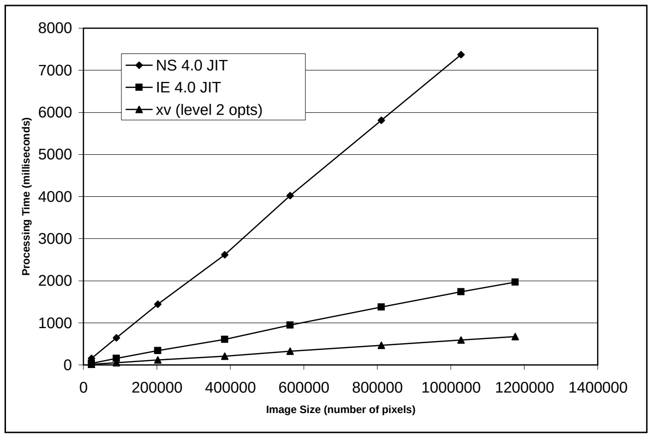

Figure 14. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Spread Image Filter to an Image at Various Sizes

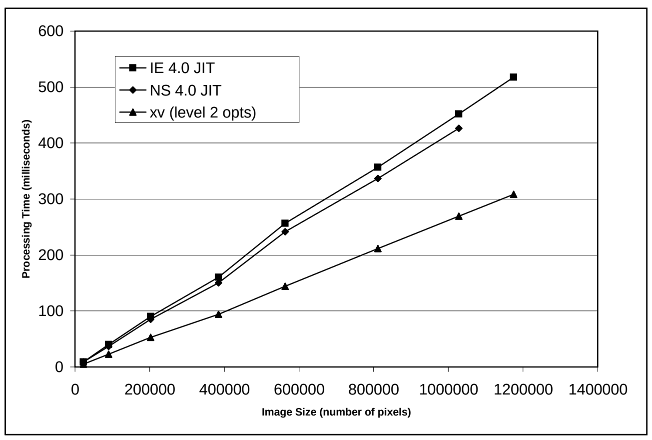

Figure 15. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Emboss Image Filter to an Image at Various Sizes

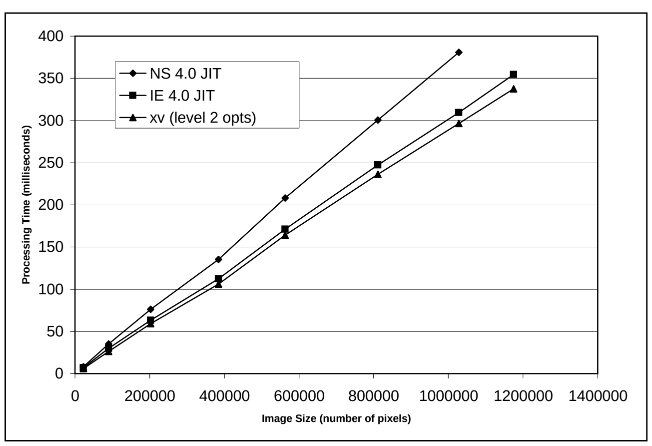

Figure 16. Comparative Performance of the Java-enabled Browsers and `xv` in Applying the Pixelize Image Filter to an Image at Various Sizes

optimizations gave the C algorithms a competitive advantage. Thus, for all but two of the filters (Blur image filer and Sharpen image filter), the C algorithms outperformed their Java equivalents.

Contributing to the overall superiority of the C algorithm execution is fast array access time and increments, induced by the rigorous utilization of registers by the compiler. However, applying the techniques of code movement and strength reduction help the Java code to negate any benefits of similar compile-time optimization performed by the C/C++ compiler.

There is one severely aberrational case in which the C runtime performed more poorly than the Java ones: the sharpen filter. In the sharpen filter, color values are continually converted between the integer RGB format and the floating point HSV format. Netscape, whose floating point to integer narrowing conversion performance exceeds Internet Explorer's and C/C++'s, has the competitive advantage.

3.2. High-speed Network Benchmarking

As described earlier, high performance is one of the key forces that guides the development of a distributed EMIS. In particular, it is important that medical images be delivered to radiologists and processed in a timely manner to allow proper diagnosis of patients exams. To evaluate the performance of Java as a transport interface for exchanging large images over high-speed networks, we performed a series of network benchmarking tests over ATM. This section compares the results with the performance of C/C++ as a transport interface [22].

3.2.1. Benchmarking Configuration

Benchmarking testbed: The network benchmarking tests were conducted using a FORE systems ASX-1000 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.5.1. The ASX-1000 is a 96 Port, OC12 622 Mbs/port switch. Each UltraSparc-2 contains two 168 MHz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.5.1 TCP/IP protocol stack is implemented using the STREAMS communication framework. Each UltraSparc-2 has 256 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

Performance metrics: To evaluate the performance of Java, we developed a test suite using Java ACE. To measure the performance of C/C++ as a transport interface, we used an extended version of TTCP protocol benchmarking tool [22]. This TTCP tool measures the throughput of transferring untyped bytestream data (*i.e.*, Blobs [14]) between two hosts. We chose untyped bytestream data, since untyped bytestream traffic is representative of image pixel data, which need not be marshaled or demarshaled.

3.2.2. Benchmarking Methodology We measured throughput as a function of sender buffer size. Sender buffer size was incremented in powers of two ranging from 1 Kbytes to 128 Kbytes. The experiment was carried out ten times for each buffer size to account for variations in ATM network traffic. The throughput was then averaged over all the runs to obtain the final results.

Since Java does not allow manipulation of the socket queue size, we had to use the default socket queue size of 8 Kbytes on SunOS 5.5. We used this socket queue size for both the Java and the C/C++ network benchmarking tests.

3.2.3. Performance Results and Evaluation

Throughput measurements: Figure 17 shows the throughput measurements using Java and C/C++ as the transport interface. These results illustrate that the C/C++ transport interfaces consistently out-perform the Java transport interfaces. The performance of both the Java version and the C/C++ version peak at the sender buffer size of 8 Kbytes. This result stems from the fact that 8 Kbytes is close to the MTU size of the ATM network, which is 9,180 bytes. The results indicate that for a sender buffer size of 1 Kbytes, C/C++ out-performs Java by only about 2%. On the other hand, for sender buffer size of 2 Kbytes, C/C++ out-performs Java by more than 50%. C/C++ out-performs Java by 15%-20% for the remaining sender buffer sizes.

Analysis summary: C/C++ out-performed Java as the transport interface for all sender buffer sizes. The difference in the performance between Java and C/C++ is reflective of the overhead

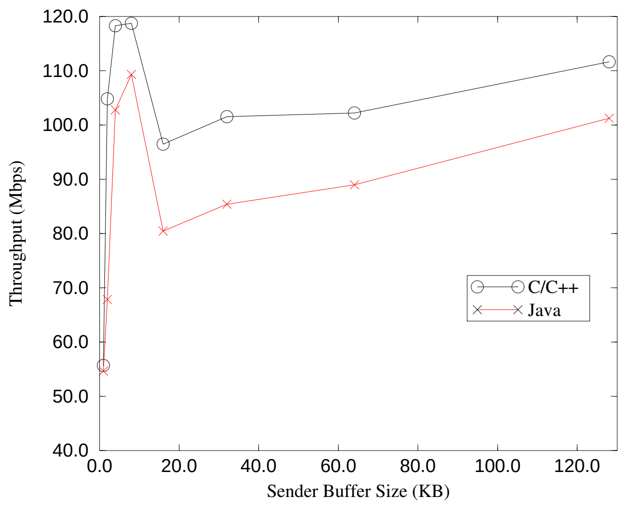

Figure 17. Throughput Measurement of Using Java and C/C++ as the Transport Interface

incurred by the JVM. This overhead can be either in the form of interpreting Java byte code (if an interpreter is used) or in the form of compiling Java byte code at run time (if a JIT compiler is used).

However, it is important to note that despite the differences in performance between Java and C/C++, Java performs comparably well. A throughput of about 110 Mbps on a 155 Mbps ATM network is quite efficient considering the default socket queue size is only 8 Kbytes. Results [23] show that network performance can improve significantly if the maximum socket queue size (64 Kbytes) is used. If Java allowed programmers to change the socket queue size the throughput should be higher for larger sender buffer sizes.

4. Related Work

Several studies have measured the performance of Java relative to other languages. In addition, many techniques have been proposed to improve the performance of Java. The following is a summary of the related work in this area.

4.1. Measuring Java's Performance

Several studies have compared the execution time of Java interpreted code and Java compiled code with the execution time of C/C++ compiled code. Shiffman [24] has measured and compared the performance of several programs written both in Java and in C++. For the tests performed, Java interpreted code performed 6 to 20 times slower than compiled C++ code, while Java compiled code performed only about 1.1 to 1.5 times slower than C++ code.

The results obtained in [24] differ from the ones we obtained because the tests run were also different. The tests carried out by Shiffman involved measuring the timings for iterative and recursive versions of a calculator of numbers in the Fibonacci series, as well as a calculator of prime numbers. The results, however, once again indicate that the Java code performs reasonably well, compared with C/C++ code. This finding is consistent with our results for the `xv` image processing algorithms.

4.2. Improving Java's Performance

Several groups are working on improving the performance of JIT compilers, as well as developing alternatives to JIT compilers.

Toba: A system for generating efficient stand-alone Java applications has been developed at the University of Arizona [25]. The system is called Toba and generates executables that are 1.5 to 4.4 times faster than alternative JVM implementations. Toba is a "Way-Ahead-of-Time" compiler and therefore converts Java code into machine code before the application is run. It translates Java class files into C code and then compiles the C code into machine code making several optimizations in the process. Although such a compiler can be very useful for stand-alone Java applications, it can not, unfortunately, be used for Java applets.

Harrisa: An efficient environment for the execution of Java programs called Harissa has been developed at the University of Rennes [26]. Harissa mixes compiled and interpreted code. It translates Java bytecode to C and in the process makes several optimizations. The resulting C code produced by Harissa is up to 140 times faster than the JDK interpreter and 30% faster than the Toba compiler described above.

Unlike Toba, Harissa can work with Java applets also. Therefore, Harissa can be used by MedJava to improve the performance of image processing and bringing it closer to the performance of a similar application written in C/C++.

Asymetrix: Another approach similar to Harissa is SuperCede VM developed by Asymetrix [27]. SuperCede is a high-performance JVM that can improve the performance of Java to execute at native C/C++ speed. Unlike JIT compilers, where the interpreter selectively compiles functions, SuperCede compiles all class files as they are downloaded from the server. The result is an application that is fully compiled to machine code and can therefore execute at native C/C++ speed. SuperCede VM can also work with Java applets and can therefore be used by MedJava to improve its performance.

4.3. Evaluating Web Browsers Performance

Several studies compare the performance of different Web browsers.

CaffeineMark: Pendragon Software [28] provides a tool called CaffeineMark that can be used for comparing different Java virtual machines on a single system, *i.e.,* comparing appletviewers, interpreters and JIT compilers from different vendors. The CaffeineMark benchmarks measures Java applet/application performance across different platforms. CaffeineMark benchmarks found Internet Explorer 3.01 on NT to contain the fastest JVM followed by Internet Explorer 3.0 on NT. Netscape Navigator 3.01 on NT performed sixth in their tests. Unfortunately, the CaffeineMark benchmarks do not include the latest versions of the Web browsers that we used to run our tests, *i.e.*, Internet Explorer 4.0 and Netscape 4.0. Therefore, their results are out-of-date.

PC Magazine: Java performance tests in PC Magazine reveal the strengths and flaws of several of today's Java environments [29]. Their tests reveal significant performance differences between Web browsers. In all their tests, browsers with JIT compilers out-perform browsers without JIT compilers by up to 20 times. This is consistent with the results we obtained. Their tests found Internet Explorer 3.0 to be the fastest Java environment currently available. They found Netscape Navigator 3.0 to be consistently slower than Internet Explorer. Once again their tests did not make use of the latest versions of the Web browsers and therefore are out-of-date.

5. Concluding Remarks

This paper describes the design and performance of a distributed electronic medical imaging system (EMIS) called MedJava that we developed using Java applets and Web technology. MedJava allows users to download images across the network and process these images.

Once an image has been processed, it can be uploaded to the server where the applet was downloaded.

The paper presents the results of systematic performance measurements of our MedJava applet. MedJava was run in two widely-used Web browsers (Netscape and Internet Explorer) and the results were compared with the performance of `xv`, which is an image processing application written in C. In addition, the paper presented performance benchmarks of using Java as a transport interface to transfer large images over high-speed ATM networks.

The following is a summary of the lessons learned while developing MedJava:

Compiled Java code performs relatively well for image processing compared to compiled C code: In our image processing tests, interpreted Java code was substantially out-performed by compiled Java code and compiled C code. The image processing application written in C out-performed MedJava in most of our tests. However, only when the C code was itself hand-optimized, were the MSVC++ compiler's compile-time optimizations able to produce significantly more efficient code. If techniques become available to employ such optimization techniques in JIT compilers without incurring unacceptable latency, then this advantage will be abated. In addition, efficient Java environments like Harissa, which mix byte code and compiled code, can further improve the performance of Java code and allow it to perform as well as the performance of C code.

Compiled Java code performs relatively well as a network transport interface compared to compiled C/C++ code: Our network benchmarks illustrate that using C/C++ as the transport interface out-performs using Java as the transport interface by 2% to 50%. The difference of 50% in performance between Java and C/C++ for a buffer size of 2 KB occurs because of a sudden jump in the throughput in the case of C/C++ in going from a sender buffer size of 1 KB to a sender buffer size of 2 KB. In the case of C/C++, throughput jumped from 55.69 Mbps to 104.81 Mbps in going from a sender buffer size of 1 KB to a sender buffer size of 2 KB. In the case of Java, however, the increase in throughput was gradual and therefore resulted in a large performance difference for sender buffer size of 2 KB.

The performance of using Java as the transport interface peaks at the sender buffer size close to the network MTU size and is only 9% slower than the performance of using C/C++ as the transport interface. Therefore, Java is relatively well-suited to be used as the transport interface.

It is becoming feasible to develop performance-sensitive distributed EMIS applications in Java: The built-in support for GUI development, the support for image processing, the support for sockets and threads, automatic memory management, and exception handling in Java simplified our task of developing MedJava. In addition, the availability of JIT compilers allowed MedJava to perform relatively well compared to a applications written in C/C++.

Therefore, we believe that it is becoming feasible to use Java to develop performance-sensitive distributed EMIS applications. In particular, even when Java code does not run quite as fast as compiled C/C++ code, it can still be a valuable tool for building distributed EMISs because it facilitates rapid prototyping and development of portable and robust applications.

Java has several limitations that must be fixed to develop production distributed EMIS: Even though Java resolves several of the forces of developing a distributed EMIS, it still has the following limitations:

• *Memory limitations:* We found that applying image filters to images larger than 1 MB causes the JVM of both Netscape 4.0 on NT and Internet Explorer 4.0 on NT to run out of memory. This can hinder the development of distributed EMISs since many medical images are larger than 1 MB.

• *Lack of AWT portability:* We found the AWT implementations across platforms to be inconsistent, thereby making it hard to develop a uniform GUI. When we tried running MedJava on different brower platforms, we found some features of MedJava do not work portably on certain platforms due to lack of support in the JVM where the applet was run.

• *Security impediments:* We found the lack of ability to upload images to servers other than the one from where the applet was downloaded from as another significant limitation of using Java for distributed EMISs. Although these restrictures were added to Java as a security-feature of applets, they can be quite limiting. The following are several workarounds for these security restrictions:

- One approach is to run a CGI Gateway at the server from where the MedJava applet is downloaded. MedJava can then make uploading requests to the Gateway that can then upload images to servers across the network.

- Another scheme can be used to solve this problem without requiring an additional Gateway to run at the server. This requires adding a security authentication mechanism within the Java `Applet` class. This mechanism can then allow an applet to upload files to servers other than the one from where the applet was downloaded. Java version 1.1 allows an applet context to download signed Java archive files (JARs), which contain Java classes, images, and sounds. If these are signed by a trusted entity using its private key, applets can run in the context with the full privileges of local applications. The context uses the public key for a entity, authenticated by a Certificate from another trusted entity, to verify that the archive file came from a trusted signer. Therefore, an EMIS signed by a trusted entity could run in a browser with the ability to save files to the local file system and open network connections to machines other than the one from which it was downloaded.

In summary, our experience suggests that Java can be very effective in developing a distributed EMIS. It is simple, portable and distributed. In addition, compiled Java code can be quite efficient. If the current limitations of Java are resolved and highly-optimizing Java compilers become available, it should be feasible to develop performance-sensitive distributed applications in Java.

The complete source code for Java ACE is available at `www.cs.wustl.edu/~schmidt/JACE.html`.

Acknowledgments

We would like to thank Karlheinz Dorn and the other members of the Siemens Medical Engineering group in Erlangen, Germany for their technical support, sponsorship, and friendship during the MedJava project.

[1] W. L. R. J. J. Conklin, “Digital Management and Regulatory Submission of Medical Images from Clinical Trials: Role and Benefits of the Core Laboratory,” *Proc. SPIE, Health Care Technology Policy II: The Role of Technology in the Cost of Health Care: Providing the Solutions*, vol. 2499, Oct. 1995.

[2] G. Blaine, M. Boyd, and S. Crider, “Project Spectrum: Scalable Bandwidth for the BJC Health System,” *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[3] I. Pyarali, T. H. Harrison, and D. C. Schmidt, “Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging,” *USENIX Computing Systems*, vol. 9, November/December 1996.

[4] F. L. Kitson, “Multimedia, Visual Computing, and the Information Superhighway,” *Proc. SPIE, Medical Imaging 1996: Image Display*, vol. 2707, Apr. 1996.

[5] P. Jain and D. Schmidt, "Experiences Converting a C++ Communication Software Framework to Java," *C++ Report*, vol. 9, January 1997.

[6] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the* 3^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, June 1997.

[7] A. Wollrath, R. Riggs, and J. Waldo, "A Distributed Object Model for the Java System," *USENIX Computing Systems*, vol. 9, November/December 1996.

[8] *Java API Documentation Version 1.0.2*. java.sun.com:80/products/jdk/1.0.2/api.

[9] M. P. Plezbert and R. Cytron, "Does Just in Time = Better Late than Never?," in *ACM 1997 Symposium on the Principles of Programming Languages*, 1997.

[10] H. R. Myler and A. R. Weeks, *Computer Imaging Recipes in C*. Prentice Hall, Inc. Englewoods Cliffs, New Jersey, 1993.

[11] J. Hu, S. Mungee, and D. C. Schmidt, "Principles for Developing and Measuring High-performance Web Servers over ATM," in *Proceeedings of INFOCOM '98*, March/April 1998.

[12] J. Hu, I. Pyarali, and D. C. Schmidt, "Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks," in *Proceedings of the* 2^{nd} *Global Internet Conference*, IEEE, November 1997.

[13] *Java ACE Home Page*. www.cs.wustl.edu/ schmidt/JACE.html.

[14] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[15] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[16] D. C. Schmidt, "Acceptor and Connector: Design Patterns for Initializing Communication Services," in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, MA: Addison-Wesley, 1997.

[17] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), Reading, MA: Addison-Wesley, 1996.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons, 1996.

[19] D. Bell, "Make Java fast: Optimize!." JavaWorld, April 1997. www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html.

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

[21] J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*. Reading, MA: Addison-Wesley, 1996.

[22] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[23] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[24] H. Shiffman, "Boosting Java Performance: Native Code & JIT Compilers." reality.sgi.com/shiffman/Java-JIT.html, 1996.

[25] T. A. Proebsting, G. Townsend, P. Bridges, J. H. H. T. Newsham, and S. Watterson, "Toba: Java For Applications, A Way Ahead of Time (WAT) Compiler," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[26] G. Muller, B. Moura, F. Bellard, and C. Consel, "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[27] Asymmetrix, "SuperCede." www.asymetrix.com/products/supercede/index.html, 1997.

[28] P. Software, "CaffeineMark(tm) 2.5: The Industry Standard Java Benchmark." www.webfayre.com/pendragon/cm2/index.html, 1996.

[29] R. V. Dragan and L. Seltzer, "Java Speed Trials." www8.zdnet.com/pcmag/features/pctech/1518/java.htm, 1996.