DESIGN AND IMPLEMENTATION OF A JAVA-BASED MPEG-1 VIDEO DECODER

Dohyung Kim
Department of Computer Engineering
Seoul National University
Seoul, Korea, 151-742
dhkim@iris.snu.ac.kr

Jihong Kim
Department of Computer Science
Seoul National University
Seoul, Korea 151-742
jihong@davinci.snu.ac.kr

Abstract

Although Java has many useful programming language features for developing consumer multimedia applications, it is not widely used for multimedia application development. One of the main reasons for the lack of Java usage in consumer multimedia application development is a concern for the execution speed of Java-based applications. In this paper, we investigate the feasibility of using Java as a language for multimedia applications. As a specific multimedia application example, we have developed a Java-based MPEG-1 video decoder. We describe the design and implementation of the MPEG-1 video decoder and report our experience in optimizing the decoding performance. Based on the performance analysis results from a Java performance profiler, we have applied both general and Java-specific optimization techniques. The final implementation could decode about 28.67 frames per second on a Pentium-II 300MHz computer for a 240 × 170 MPEG-1 video bitstream, a speed-up of 2.8 times over the initial implementation. Our experience strongly suggests that the pure Java-based media processing is a feasible solution.

Keywords: Java performance, MPEG-1 video decoder, performance evaluation.

1 Introduction

Java has many useful programming language features for developing consumer multimedia applications. For example, the built-in support for the concurrency and object-oriented design can help a programmer in designing and implementing a complex multimedia application such as a video decoder. In addition, Java's platform independence (write once, run everywhere) feature is very attractive for multi-platform application developers.

In spite of these advantages, however, Java has not been widely used for multimedia application development. One of the main reasons for the lack of Java usage in multimedia application development is a concern for the execution speed of Java-based applications due to a Java's interpretation-based execution model. In this paper, we investigate the possibility of using Java for multimedia applications. As a specific example application, we have developed a Java-based MPEG-1 video decoder and evaluated if it can decode video bitstreams in a real-time rate, 30 frames per second. We describe the design and implementation of the Java MPEG-1 video decoder and report our experience in evaluating and optimizing the performance of the Java MPEG-1 video decoder.

Our work has a similar goal as the work done by Patel, Smith and Rowe for a C-based MPEG-1 video decoder [1], in that the both works evaluate the performance of a software-only MPEG-1 video decoder and explore the possibility of supporting the real-time decoding by a software-only MPEG-1 video decoder. Anders had implemented a Java-based MPEG-1 video decoder but its performance was far from the real-time decoding [2].

The remainder of this paper is organized as follows. Section 2 presents a brief introduction to the MPEG-1 video decoding process. Section 3 describes the overall implementation and functions of the Java MPEG-1 video decoder. Section 4 gives a detailed description of the optimization techniques used and presents the various experimental results. We conclude with a summary in Section 5.

2 MPEG-1 Video Decoding Process

In this section, we briefly describe the MPEG-1 video decoding process [3]. (For a detailed description of the MPEG-1 video compression standard, refer to various reference books such as [4]). Since a decoding process is the reverse of an encoding process, we explain the MPEG-1 video compression first.

In order to compress a sequence of images, MPEG-1 video coding uses three techniques, *transform coding*, *motion compensation* and *entropy coding*. Transform coding exploits the characteristics of human visual system and concentrates the energy of an image to fewer values by a mathematical transform. Like many other

Contributed Paper
Manuscript received August 9, 1999

image/video compression standards, MPEG-1 video coding uses an 8 × 8 discrete cosine transform (DCT) for this purpose. DCT converts an 8 × 8 image block from the spatial domain to the DCT domain.

Before applying DCT, MPEG-1 video coding converts RGB images to YCrCb images. Y value indicates the luminance level and Cr/Cb values represent chrominance. Since the human eye is more sensitive to luminance than to chrominance, Cr/Cb values are subsampled. A YCrCb image is divided into macroblocks. Each macroblock represents a 16 × 16 pixel area (i.e., four 8 × 8 blocks) of Y image and one 8 × 8 block from each of Cr/Cb images. Each block is then transformed by DCT and quantized. Quantization is the only *lossy* compression step used in MPEG-1 video coding and achieves the compression by removing the high frequency information to which human visual system is less sensitive. To further compress video data, entropy coding techniques such as Huffman coding and run-length encoding are used.

Motion compensation exploits the similarity between successive images. Large compression can be accomplished by encoding only the differences between images, instead of images themselves. For the motion compensation purpose, MPEG-1 video coding classifies images into one of three picture types - I (intracoded) picture, P (predicted) picture and B (bidirectional) picture. I picture uses only transform coding, but P and B pictures use both motion compensation and transform coding. In P and B pictures, for each macroblock, the best corresponding match in the reference image is found. Once the best match is found, the error macroblock is computed by subtracting the current image from the reference image pixel by pixel. DCT is then applied to the error macroblock. After the motion compensation and transform coding are performed, entropy coding is used to further compress bitstreams.

Figure 1 shows the overall steps of the MPEG-1 video decoding process. First, a decoder reads a packet and extracts video and header information. By Huffman decoding, DCT values are computed. In case of I pictures, the decoder converts DCT values to YCrCb values directly by applying IDCT function. In case of P and B pictures, DCT values are the difference values between the reference image and the currently reconstructed image, and IDCT reconstructs the difference image. Using a motion information for a macroblock, the decoder reconstructs the current image by adding the difference image to the reference image. B pictures are decoded in a similar fashion as P pictures except that two reference pictures are used, one from the proceeding pictures, and the other from the succeeding pictures. Once YCrCb pictures are reconstructed, they are converted to the RGB format for display.

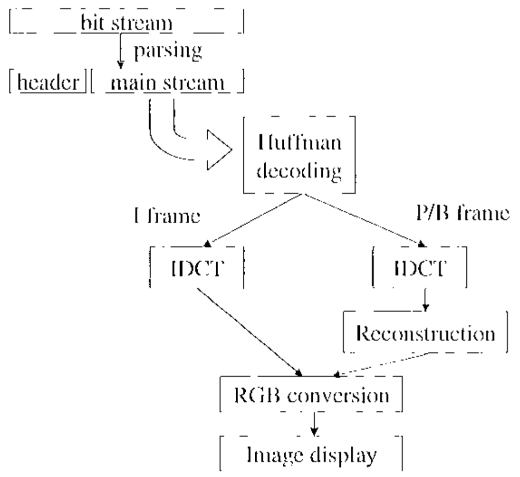

Figure 1. MPEG-1 video decoding process.

3 Overview of Java MPEG-1 Video Decoder Implementation

Our performance goal in developing a Java-based MPEG-1 video decoder was to support a real time decoding that can display 30 frames per second. In this section, we explain the design and implementation of the first version of our Java MPEG-1 video decoder.

The Java MPEG-1 video decoder consists of six major functional modules: the bitstream reader module, the stream parser module, the IDCT module, the reconstruction module, the RGB conversion module and the image display module. The bitstream reader module supports the manipulation of bit values. Since Java does not have the bit data type as a first-class citizen, we made a new class for processing bitstreams. The new class extends the buffered IO class for efficiency. The stream parser module extracts headers from a bitstream and decodes entropy-coded bitstreams. As with the Berkeley MPEG-1 video decoder [1], the Huffman decoding is implemented using a hierarchical lookup-table technique. The IDCT module transforms an 8 × 8 block in the DCT domain to an 8 × 8 block in the spatial domain. For the computational efficiency, 8 × 8 2D IDCT was implemented by using two 1 × 8 1D IDCTs. We used Lee's algorithm for 1 × 8 IDCT [5].

The reconstruction module rebuilds pictures based on the motion information and reference pictures. The RGB conversion module converts YCrCb values to RGB values for display. In converting one YCrCb triplet to a RGB

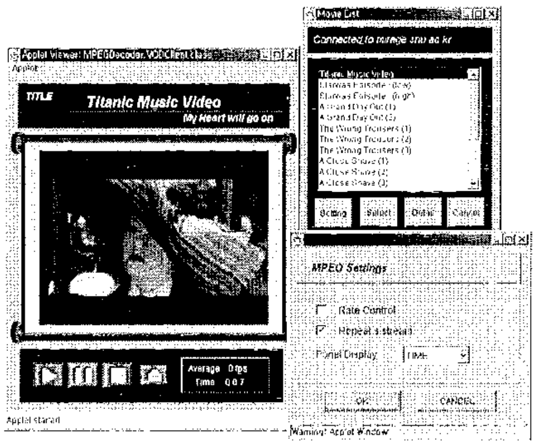

Figure 2. Snapshot of the Java MPEG-1 video decoder.

triplet, 11 floating-point additions and 7 floating-point multiplications are necessary.

Figure 2 shows a snapshot of our Java MPEG-1 video decoder. There are three user interface windows: the display window (the left window in Figure 2), the movie list window (the upper right window in Figure 2) and the parameter setting window (the lower right window in Figure 2). The user selects a movie through the movie list window and sets decoding related parameters (e.g., rate control) through the parameter setting window. Once started, the selected movie is displayed in the display window. The major functions of the video decoder are summarized in Figure 3. (For interested readers, we recommend to visit our video decoder homepage at http://mirage.snu.ac.kr/dhkim/java/MPEG.)

4 Optimizations and Experimental Results

The performance of the first version of our video decoder was rather disappointing. For the two test MPEG-1 video bitstreams listed in Table 1, our decoder can play only 10.42 frames per second on a Pentium II 300MHz machine. In order to understand the performance bottlenecks of the first implementation, we have used a Java profiling tool [6]. Table 2 shows the performance analysis results for the test bitstreams. As shown in Table 2, we have identified three modules as the main performance bottlenecks: the IDCT module (IDCT), the RGB conversion module (RGB) and the reconstruction module (Recon). Although the image display modules (Raster and Display) also consumed a fair amount of execution cycles, we did not consider them for the further optimization because they heavily depend on the performance of Java API routines.

1. Display a movie title and a subtitle.
2. Play, pause and resume a movie.
3. Adjust a display screen to match the movie size.
4. Display the average number of frames decoded per second.
5. Adjust the rate control parameters.
6. Replay the same movie continuously if selected.
7. Adjust play time.

Figure 3. Major functions of the Java MPEG-1 video decoder.

Stream	Size	FPS	BPS	I:P:B
Movie 1	320 × 240	24	786432	1:5:5
Movie 2	240 × 176	30	598016	2:1:0

Table 1. Test MPEG-1 video bitstreams.

	Movie 1		Movie 2	
Module	%	Time (ms)	%	Time (ms)
IDCT	43.22%	204041	44.88%	97593
RGB	19.13%	90358	20.53%	44663
Recon	8.99%	42480	2.03%	4424
Raster	5.81%	27463	6.01%	13794
Display	4.83%	22828	4.92%	10714

Table 2. Performance analysis of the first implementation.

In order to improve the performance of the Java MPEG-1 video decoder, we have used two categories of optimization techniques: *general optimization techniques* that are widely used for general programming works, and *Java-specific optimization techniques* that reduce the performance overhead of using Java-related language features.

4.1 General Optimization Techniques

The optimization techniques in this category generally reduce the computational requirements. The representative techniques are fixed-point arithmetic, look-up table-based multiplication, loop unrolling, etc.

The fixed-point arithmetic technique was used to optimize the RGB-to-YCrCb conversion module. In order to convert a RGB triplet to a YCrCb triplet based on a straightforward implementation, it takes 7 floating-point multiplications and 11 floating-point additions. If the image resolution is 352 × 240, about 600,000 floating-point multiplications and 930,000 floating-point additions are necessary. In the optimized RGB-to-YCrCb conversion implementation, we have replaced floating-point multiplications with integer multiplications and shift operations. In order to further reduce the number of arithmetic operations, we implemented integer multiplications based on a look-up table technique. We also unrolled the conversion loop four times to reduce the loop overhead and used inline functions to reduce the subroutine call overhead. These optimizations reduced the number of operations per pixel from 18 operations to 12.8 operations.

For the implementation of Lee's IDCT algorithm, we have replaced floating-point multiplications with fixed-point multiplications as well. In order to take advantages of the characteristics of the compressed bitstreams, in implementing the 8 × 8 2D IDCT by two 1D IDCTs, we checked if all eight coefficients are zeroes or not. If they are all zeroes, we skipped the 1D IDCT calculations. Since the overhead of checking whether the coefficients are all zeroes are not insignificant, this test was performed for the first 1D IDCT only.

In order to understand the performance gains from using the fixed-point arithmetic in Java, we measured the cost of basic Java operations using a Java benchmark on a Pentium II 300MHz machine [7]. Table 3 summarizes the measurements. Surprisingly, the results showed that there were no significant performance differences between the basic floating-point arithmetic operations and integer arithmetic operations. Furthermore, for a multiplication operation, the measurements showed that a floating-point arithmetic operation is even *faster* than an integer operation, suggesting that we had better use floating-point multiplications instead of integer multiplications and shifts. However, the dominating penalty of using the floating-point arithmetic does not come from arithmetic

Operation	Time (ps)
int = (int) float	180155
float = (float) int	6821
int = int + int	4714
int = int * int	5842
int = int >> int	3288
int = int | int	4499
float = float + float	4150
float = float * float	3281
new Object()	818683

Table 3. Measured cost of basic Java operations.

operations but type casting operations. As shown in Table 3, a float-to-integer type cast operation takes over 50 times more time than a floating-point multiplication operation. In the floating-point implementation, type casting operations dominate the cost of an IDCT implementation.

For the reconstruction module, we focused on reducing the loop overhead and control overhead. Unrolling the reconstruction loop body several times reduced the loop overhead, while making separate subroutines for various macroblock types minimized the control overhead. Although some functions were duplicated, making a separate function for each different macroblock type removes the repeated if-then-else checks.

Figure 4 shows the performance improvements by the general optimization techniques using the Movie 2 test bitstream. In the RGB module, the processing time was reduced by 22.4% using the fixed-point arithmetic over the initial implementation. The look-up table-based multiplications further improved the performance by 68.8% over the implementation based on the fixed-point arithmetic. For the IDCT implementation, the fixed-point multiplication method plus a 1D IDCT skip technique reduced the processing time by 34.8% over the first version.

4.2 Java Specific Optimization Techniques

As described in [8], one of the main sources of Java performance problems comes from the high cost of object creations and object-to-object copy operations. (This can be easily verified from Table 3: the object creation is almost 200 times more expensive than a multiplication operation.) A liberal use of Java objects is not only expensive during the object creation time but also detrimental to the performance of a garbage collector, resulting in an overall poor performance. In order to

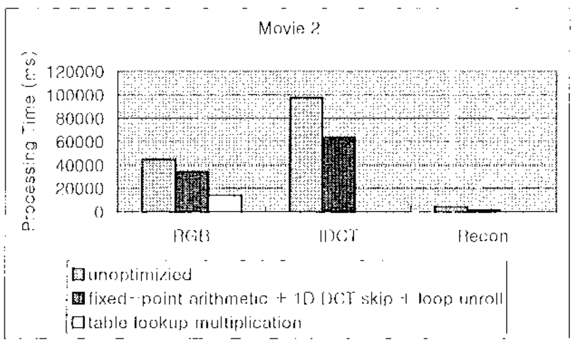

Figure 4. Performance improvements by the general optimization techniques.

reduce the object-related performance overhead, we have applied the object reuse (or object pooling) technique.

The object pooling technique is very effective for the Java MPEG-1 video decoder because the decoding process requires many large objects such as frames (e.g., the current frame and reference frames) and display buffers. In our optimization, instead of dynamically recreating these objects whenever necessary, we reused them without recreating once they are created in the initial decoding stage.

The second optimization was to avoid the usage of array objects as much as possible. Since a video decoder manipulates a sequential bitstream, an array is a convenient data structure. However, in Java, an array reference is an expensive operation. For example, for each array reference, the bound check should be performed to guarantee the array index has a valid value. For time-critical functions where the number of array elements is known as a constant, we did not use an array but the same number of scalar variables. For example, 1D 1 × 8 IDCT was a good candidate for this type of optimization because the number of elements are fixed as 8.

In addition to the object pooling technique, if possible, functions were declared as final or static. This enables a compiler to do inline function expansions and to support fast function calls by an underlying JVM. We also found that collapsing several equations into one equation improves the Java performance significantly. Unless the result of an equation is reused in the later computation, we collapsed the multiple equations into one equation.

Figure 5 shows the performance improvement of the IDCT module by applying the Java specific optimization techniques. For the Movie 2 test bitstream, we can see that the performance was improved more than 300% by not using arrays and collapsing equations.

4.3 Experimental Results

We have conducted some experiments to examine the overall performance improvements by the techniques described in Sections 4.1 and 4.2. For this purpose, we have used three versions of Java MPEG-1 video decoder described in Table 4. Three video decoder programs differ in the optimization techniques used. Program 1 is the initial implementation without using any optimization techniques described in Sections 4.1 and 4.2 while Program 3 uses all the optimization techniques. Table 5 lists the decoding performance in terms of the average number of frames decoded per second (average fps) for the two test bitstreams. As can be seen in Table 5, the

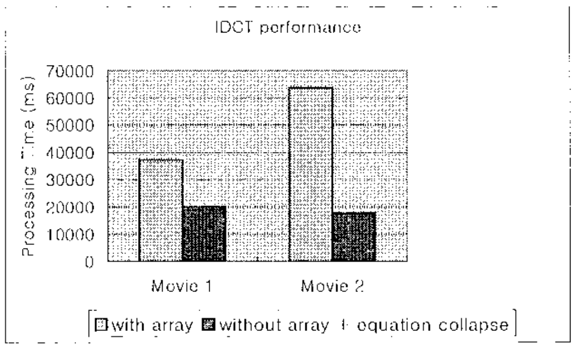

Figure 5. Performance improvement of the IDCT implementation using the Java specific optimization techniques.

decoding performance is improved by up to a factor of 3.6.

Figures 6 and 7 show the overall distribution of processing times for the three decoder programs. The optimization techniques described in Sections 4.1 and 4.2 were effective for the IDCT, RGB and reconstruction modules. The IDCT module was improved by 1000% and 550%, the RGB conversion by 300% and 320%, and the reconstruction module by 500% and 270% for Movie 1 and Movie2, respectively. For the rasterization and display modules, however, the optimization techniques described were not applicable because these two modules heavily depends on the performance of Java API implementations, which we could not optimize much.

In order to validate that the performance improvements by the described optimization techniques are not limited to the test bitstreams and the test machine, we measured the performance of the Java MPEG-1 video decoder using

Program	Optimization Techniques Used
Program 1	No optimizations used.
Program 2	Use static/final methods. Use integer multiplications and shifts. Skip 1D IDCT if all elements are zeroes. Use loop unrolling in reconstruction.
Program 3	Techniques used for Program 2. Reuse objects. Collapse equations. Avoid using arrays. Use a lookup table in RGB conversion.

Table 4. Three versions of Java MPEG-1 video decoder.

	Movie 1	Movie 2
Program 1	4.22 fps	10.42 fps
Program 2	12.00 fps	18.00 fps
Program 3	15.07 fps	28.67 fps

Table 5. Decoding rate for three versions of Java MPEG-1 video decoder.

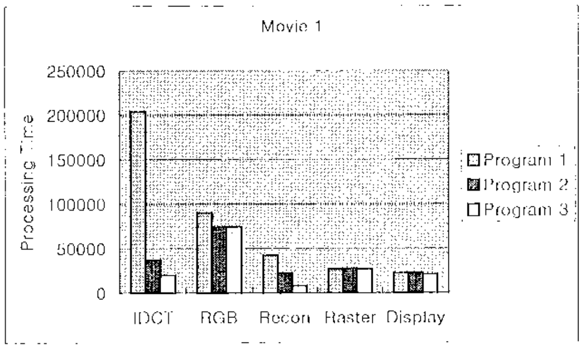

Figure 6. Processing time distribution for Movie 1 using three versions of the MPEG-1 video decoder (time in ms).

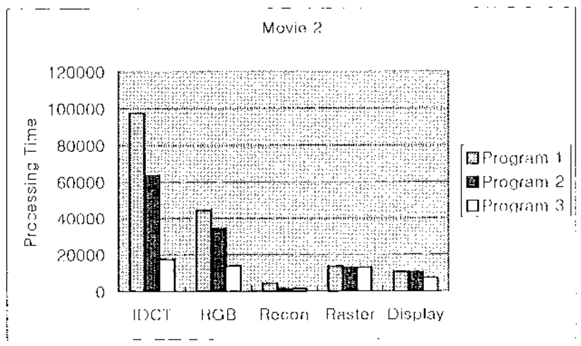

Figure 7. Processing time distribution for Movie 2 using three versions of the MPEG-1 video decoder (time in ms).

Stream	Size	FPS	Bit Rate	I:P:B
Bitstream 1	320 × 240	24	96K	1:5:5
Bitstream 2	320 × 144	24	73K	1:4:10
Bitstream 3	304 × 224	30	73K	2:1:0
Bitstream 4	240 × 176	30	73K	2:1:0
Bitstream 5	240 × 108	24	54K	1:4:10

Table 6. Test Bitstreams.

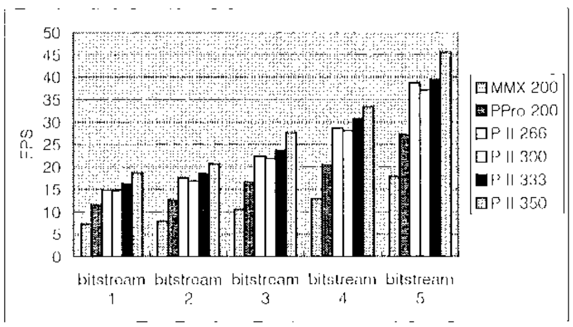

Figure 8. Java MPEG-1 video decoder performance on different machines.

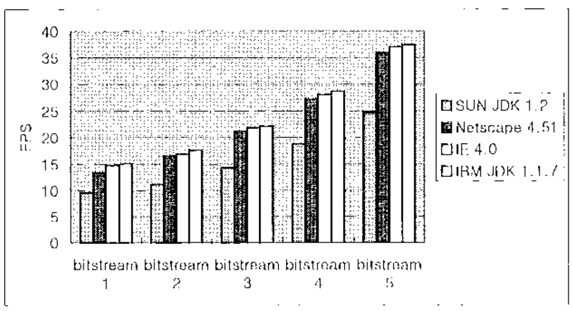

Figure 9. Java MPEG-1 video decoder performance under different JVMs on a Pentium II 300 machine.

different bitstreams, different Java virtual machines (JVMs) and different machines. Table 6 describes the characteristics of five test bitstreams tested. Figure 8 shows the performance measurements on various machines using five test bitstreams. In general, the performance improves as the processor speed increases. (The exception is the case between Pentium II 266MHz and Pentium II 300MHz. This is because the Pentium II 300MHz machine is a notebook computer. It uses the slower memory and slower bus architecture.) Figure 9 shows the performance of the Java MPEG-1 video decoder at different JVMs on a Pentium II 300MHz machine. It shows the performance can be quite different depending on which JVM is used.

5 Conclusions

In order to evaluate the feasibility of Java as a multimedia programming language, we have presented the design and implementation of a Java-based MPEG-1 video decoder and described several useful optimization techniques used to speed up the decoder.

The initial implementation of our MPEG-1 video decoder was able to play 10.42 frames per second on a Pentium-II 300 MHz computer, far short of our target rate, 30 frames per second. Based on the performance analysis results from a Java performance profiler, we have applied two categories of optimization techniques to improve the performance of the video decoder. First, we have applied general optimization techniques to compute-bound routines such as the inverse discrete cosine transform (IDCT) and color space conversion to reduce the computational requirements. Second, we have used Java specific optimizations such as the object pooling and scalar-based array replacement techniques to reduce the memory management overhead and array reference overhead in Java. After these optimizations, the MPEG-1 video decoder could decode about 28.67 frames per second on the same machine for a 240 × 170 MPEG-1 video bitstream, a speed-up of 2.8 times over the initial implementation. Based on our experience with the Java-based MPEG-1 video decoder, we strongly believe that the pure Java-based media processing is a feasible solution.

References

[1] K. Patel, B. Smith and L. Rowe, Performance of a Software MPEG Video Decoder, *Proceedings of the conference on Multimedia '93*, pp. 75-82, 1993.

[2] J. Anders, Inline MPEG-1 player in Java, *http://rnvs.informatik.tuchemnitz.de/~ja/MPEG/MPEG_Play.html*.

[3] D. Le Gall, MPEG: A Video Compression Standard for Multimedia Applications, *Communications of the ACM*, Vol. 34, No. 4, Apr. 1991, pp. 46-58.

[4] J. Mitchell, D. Le Gall, and C. Fogg, *MPEG Video Compression Standard*, Chapman and Hall, 1996.

[5] B. Lee, A New Algorithm to Compute the Discrete Cosine Transform, *IEEE Transactions on Acoustics, Speech, And Signal Processing*, Vol. ASSP-32, No. 6, Dec. 1984, pp. 1243-1245.

[6] OptimizeIt home page, http://www.optimizeit.com.

[7] G. Freedman, Java Performance Issues and Solutions, *http://www.optimizeit.com/Optimization.pdf.*

[8] R. Klemm, Practical Guidelines for Boosting Java Server Performance, *Proceedings of the ACM 1999 Java Grande Conference*, pp. 25-34, 1999.

Biographies

Dohyung Kim received BE and MS degrees in computer engineering from Seoul National University in 1997 and 1999, respectively. Currently he is working toward Ph.D. degree at the same department. His research interests are in the areas of design automation and Java performance evaluation.

Jihong Kim is an assistant professor in the Department of Computer Science, Seoul National University. Before joining Seoul National University in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. Jihong Kim received his BS in computer science and statistics from Seoul National University in 1986, and MS and Ph.D. degrees in computer science and engineering from the University of Washington in 1988 and 1995, respectively. His research interests include computer architecture, embedded systems, Java computing, multimedia systems, and real-time systems. He is a member of the IEEE Computer Society and ACM.