Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1999

Experimental Evaluation of Design Tradeoff in Specialized Virtual Machine for Multimedia Traffic in Active Networks

Sheng-Yih Wang

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
99-047

Wang, Sheng-Yih and Bhargava, Bharat, "Experimental Evaluation of Design Tradeoff in Specialized Virtual Machine for Multimedia Traffic in Active Networks" (1999). *Department of Computer Science Technical Reports.* Paper 1477.
https://docs.lib.purdue.edu/cstech/1477

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

EXPERIMENTAL EVALUATION OF DESIGN TRADEOFF IN SPECIALIZED VIRTUAL MACHNINE FOR MULTIMEDIA TRAFFIC IN ACTIVE NETWORKS

Sheng-Yih Wang
Bharat Bhargava

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #99-047
December 1999

Technical Report CS-99-047
Experimental Evaluation of Design Tradeoff in Specialized Virtual Machine for Multimedia Traffic in Active Networks*

Sheng-Yih Wang and Bharat Bhargava
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
E-mail: {swang,bb}@cs.purdue.edu

December 1999

1 Introduction

Active Networks [6] and Active Services [2] are becoming popular in recent years. In an active network environment, the packets can carry active programs within the packet to enable specialized processing on them. One of the major application of active techniques is video transcoding [3]. Since the emerging applications will contain a lot of multimedia data such as video and audio data, it is a very challenging issue on how to provide specialized multimedia data processing capabilities within these new service architectures. Current active network design such as Smart Packets [13] or ANTS [15] either focus on small data or use general purpose mobile languages and virtual machines such as Java in their architectures. In our proposed active network architecture [14], we argue that general purpose virtual machines may not be capable of handling the multimedia data and a customized virtual machine which includes specialized processing functions is needed.

In this paper, we try to quantify the effectiveness of general capsule programs v.s. specialized processing functions for multimedia data through some experiments. From these experiments, we can draw some conclusions on the trade-off between the

*This research is partly supported by a grant from NSF under NCR-9405931

general programming model and the specialized functions provided by the router for the active capsules. We also obtain some quantitative data for those trade-offs. These data can be used to guide our design of the active network architecture. These data can also be interpolated or extrapolated to predict the hardware requirements (CPU speed, memory speed/capacity, etc.) for emerging active network architectures.

The paper is organized as follows. In section 2, we describe the background of the problem and raise the issues and questions that need to be addressed when considering the design of the specialized virtual machine. In section 3, we conduct a series of experiments designed to partially answer the questions. Finally, conclusion and future works are discussed in section 4.

2 Background

2.1 Active Networks and Virtual Machines

In an active network architecture, the network is consist of a set of active or non-active nodes. Active nodes run a Node Operating System (NodeOS) and one or more Execution Environments (EEs) run on the top of NodeOS [1]. Each EE implements a virtual machine which runs the active programs coming with the data packets. The virtual machine can be very general (such as Java Virtual Machine, which is the most popular one) or very specific (such as Spanner [13] in the Smart Packet project, which provide specialized functions for network management tasks). Since emerging applications will contain a lot of multimedia data such as compressed video and audio, some specialized processing functions are necessary in order to provide better services to these data.

We have proposed a network architecture [14] which encompass active networks as one of the major component. In our architecture, we have a specialized virtual machine which will provides multimedia data processing capabilities. To design such a virtual machine, some issues need to be investigated.

2.2 Issues

To understand the design issues of the specialized virtual machine, we have to ask the following questions:

- Do general script or virtual machines fast enough to implement the complex algorithm such as MPEG video decoding? How fast can mobile code techniques achieve in comparison with the native code implementation?

- Is there any part of the video processing algorithm that can be extracted and implemented as common operations in the specialized virtual machine?

- How big is the active programs when a non-trivial algorithm such as MPEG decoder is implemented? Is it feasible to implement these algorithms using the mobile codes?

Since the above questions are very general and depends on many different factors such as the choice of virtual machine and video processing algorithm, we will investigate one special case which we feel can provide some insights into the real scenario. Specifically, we will analyze the MPEG video decoding algorithm under C and Java implementations. Since MPEG video decoding is a very essential part of compressed video processing such as transcoding [3], the data we gathered will be useful not only for MPEG algorithms but also for a broader range of multimedia applications. Please note that we are not trying to answer the general benchmark questions of Java virtual machines. There are a lot of works on this topic available such as [9, 11]. Our goal here is to get an estimate of how mobile codes can perform in the problem of multimedia data processing because it will be an essential part of our virtual machine.

3 Experiments

In the following experiments, we use three MPEG video clips which we digitized from various commercial video tapes. We believe these testing data are more representative for real-world situation than many moving-head clips created in research laboratories. They are:

- *Jurassic Park* (JP) - A typical movie with different type of scenes. The characteristics of this video clip is similar to most of the other videos in the market.
- *Tai Chi* (TC) - An instruction video which teach Tai Chi Chun (one form of Chinese Martial Art). This video is a typical instruction video. The feature of a typical instruction video is that usually there are no fast changing scenes and background. The changes in the video is slow because most of the scenes involve only motions of the instructor.
- *Lion King* (LK) - A cartoon video. The feature of a cartoon video is that there are many sharp edges which usually don't occur in real world video. The compression of cartoon is usually more difficult because MPEG scheme works better on real-world objects.

Table 1 gives the details of these video clips.

All the MPEG video clips are sampled at the rate of 30 frames/sec. The number in the parenthesis are the number of frames of I, P, and B types. For example, there are 251 I frames, 750 P frames and 1999 B frames in the video Jurassic Park. The

Clip Name	Jurassic Park	Tai Chi	Lion King
Resolution	320 x 240	352 x 240	320 x 240
# of frames	3000	2996	3750
Total Size	23091162	12497528	10273216
Avg. I frame size	19458.98 (251)	7684.57 (334)	8056.66 (313)
Avg. P frame size	14423.05 (750)	5632.38 (666)	4752.44 (938)
Avg. B frame size	3695.67 (1999)	3094.44 (1996)	1317.00 (2499)
Overall avg. frame size	7696.38	4170.34	2738.85
GOP pattern	IBBPBBPBBPBB	IBBPBBPBB	IBBPBBPBBPBB

Table 1: Profiles of Three Video Clips

average picture size is 7696 bytes, and the average size of I, P and B frames are 19459, 14423, and 3696 bytes respectively.

There are two machines used in the following experiments. One machine is an Intel Pentium II 300 MHz Personal Computer which runs both RedHat 6.0 Linux operating system and Microsoft Windows 98. The other one is a Sun Sparc 10 workstation running Solaris 2.5.

3.1 Experiment A: Comparison of Java and C implementation of MPEG video decoder

The first experiment is designed to determine if the Java virtual machines and bytecodes are fast enough to implement complex algorithms such as MPEG video decoding. It also demonstrate how fast currently available mobile code techniques can achieve in comparison with the native code implementation. To get a rough idea of how fast the mobile code techniques can achieve, we compare the decoding time of both a Java and C implementation of the MPEG-I video decoder. For C implementation of the MPEG video decoder, we use the Berkeley *mpeg_play* [12] programs. For Java implementation, we modify the decoder written by professor Joerg Anders at Technische Universität Chemnitz, Germany [4]. We have compiled the C implementation under two configurations:

1. Linux which using version 2.2 kernel and gcc C compiler with optimization.

2. Solaris 2.5 which use cc compiler with optimization.

We ran the Java implementation using Java 2 JDK from Sun Microsystems under Linux, Solaris and Microsoft Windows 98. Except in Experiment Set 3 which is used as a baseline reference only, we enable the Java Just-In-Time (JIT) compiler to speed up the bytecode execution. Table 2, 3 and 4 tabulate the number we collected.

Experiment Set	1	2	3	4	5	6
Implementation	C	C	Java2	Java2	Java2	Java2
OS	Linux	Solaris	Linux	Linux	Solaris	MS Win 98
VM/Compiler	gcc	cc	Sun	Sun/JIT	Sun/JIT	Sun/JIT
Machine	P II 300	Sparc 10	P II 300	P II 300	Sparc 10	P II 300
I frames Avg.	24.24	208.89	920.25	228.99	1258.34	61.32
P frames Avg.	20.36	178.27	897.22	200.77	1138.53	61.69
B frames Avg.	9.30	73.22	509.03	116.52	644.02	37.91
Overall Avg.	13.31	110.83	640.48	146.99	819.05	45.81

Table 2: Average decoding time for clip Jurassic Park

Experiment Set	1	2	3	4	5	6
Implementation	C	C	Java2	Java2	Java2	Java2
OS	Linux	Solaris	Linux	Linux	Solaris	MS Win 98
VM/Compiler	gcc	cc	Sun	Sun/JIT	Sun/JIT	Sun/JIT
Machine	P II 300	Sparc 10	P II 300	P II 300	Sparc 10	P II 300
I frames Avg.	16.98	123.20	584.29	194.86	604.41	38.86
P frames Avg.	14.66	101.59	557.84	196.59	620.64	42.67
B frames Avg.	10.51	66.52	363.95	123.45	404.05	27.28
Overall Avg.	12.15	80.64	431.62	147.67	474.53	31.99

Table 3: Average decoding time for clip Tai Chi

To show the relative performance of the C and Java implementations for the same platform, we plot the relative decoding times in figures 1, 2 and 3. In those figures, the experiment set 1 and 2 are normalized as 1 and the numbers of other experiment sets are relative to the corresponding C implementation of the same platform.

From the result, we made some observations:

1. The Java decoder is, in general, too slow in comparison with the C decoder. If we compare the two decoders under the same OS (Linux or Sun Solaris), the Java decoder is almost 10 times slower under Linux (data set TC) and 6 time slower under Solaris (data set JP). Although these result is not surprises to us (we know that Java decoder will be much slower than the C decoder.), it do provide an idea of how big the magnitude of slowdown it will be when using Java on some real-world problems. One interesting twist to the results is that when the same Java decoder was ran under Microsoft Windows 98, the slow down is not as big as those in the other platforms. The overall average decoding time is about 2.6 times slower under data set TC. Since *mpeg_play* is an X-Windows based program, it is not easy to port to MS Windows 98. Therefore we didn't

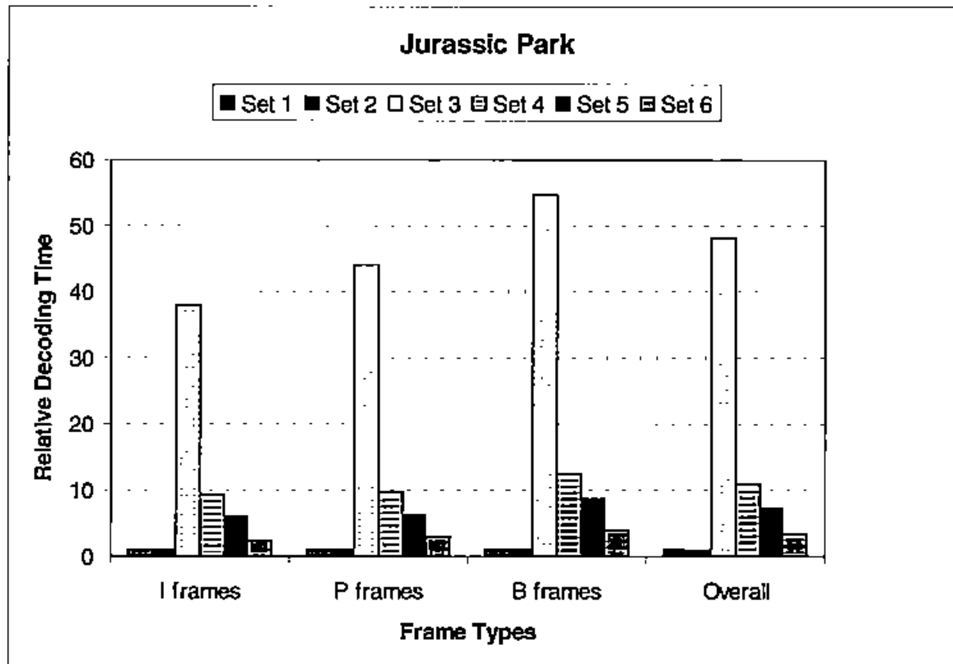

Figure 1: Average decoding time for clip Jurassic Park

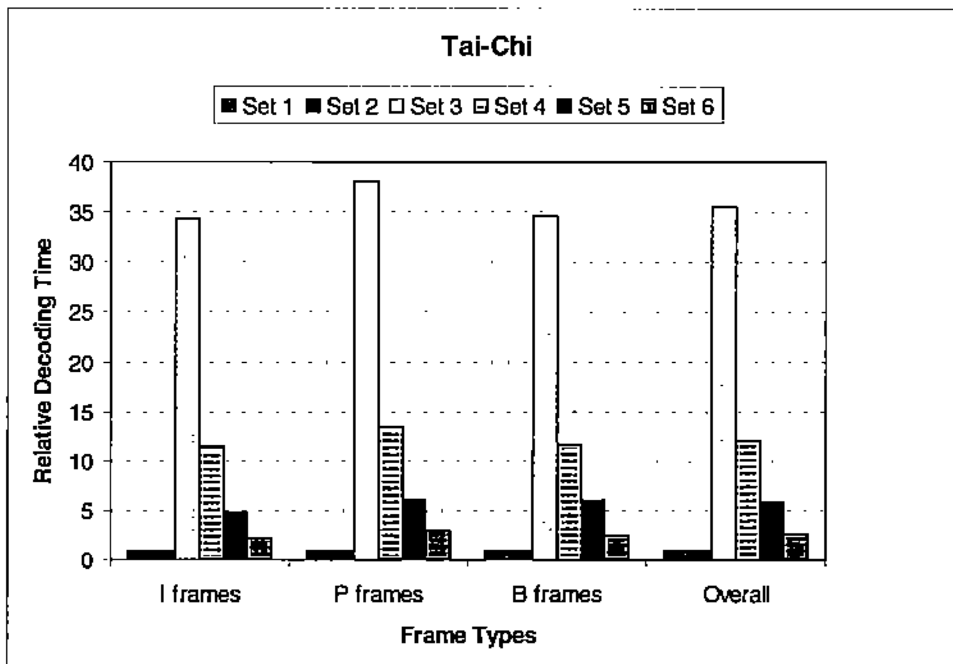

Figure 2: Average decoding time for clip Tai Chi

Experiment Set	1	2	3	4	5	6
Implementation	C	C	Java2	Java2	Java2	Java2
OS	Linux	Solaris	Linux	Linux	Solaris	MS Win 98
VM/Compiler	gcc	cc	Sun	Sun/JIT	Sun/JIT	Sun/JIT
Machine	P II 300	Sparc 10	P II 300	P II 300	Sparc 10	P II 300
I frames Avg.	17.77	124.24	557.94	141.40	568.56	38.07
P frames Avg.	11.38	78.04	426.22	96.81	432.70	33.40
B frames Avg.	5.52	33.24	369.00	64.30	328.74	30.59
Overall Avg.	8.01	52.04	399.08	78.87	374.76	31.92

Table 4: Average decoding time for clip Lion King

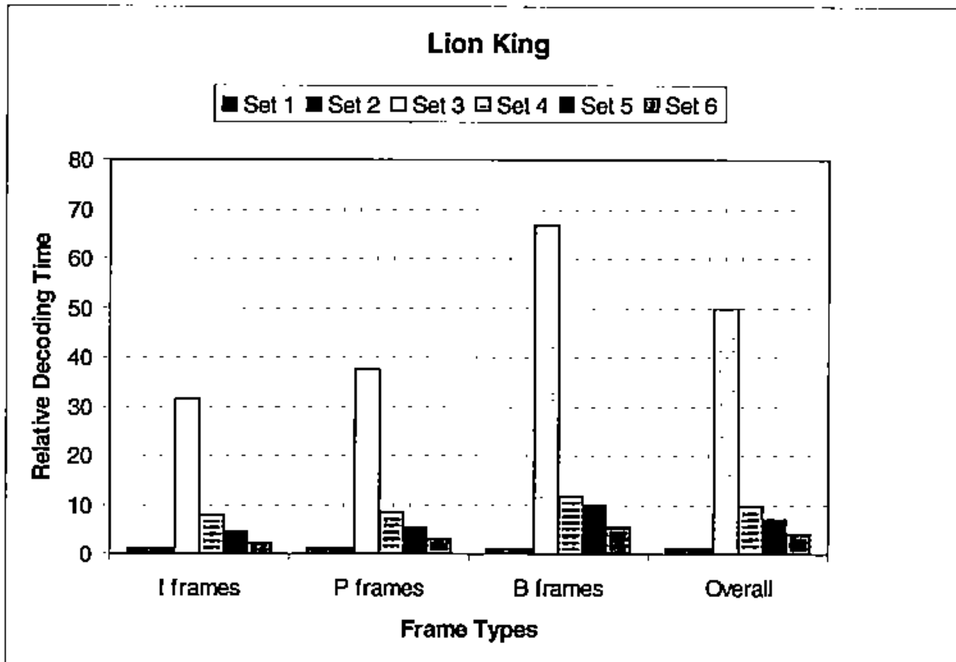

Figure 3: Average decoding time for clip Lion King

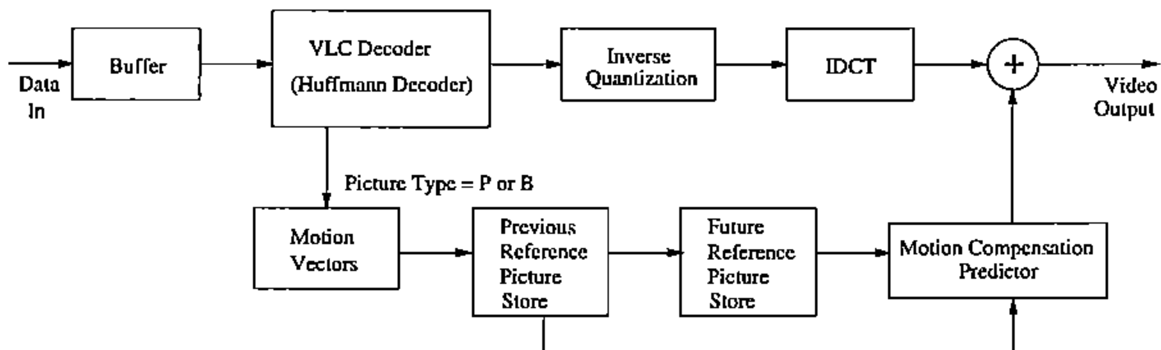

Figure 4: MPEG Decoding Process

gather the data for C decoder under MS Windows. However, we believe that the number will be very close to the number in Experiment Set 1.

2. Comparing experiments Set 3 and Set 4, it is clear that the JIT compiler do provide substantial improvement in execution time over the bytecode interpreter (over four times of speed up). However, there are still a lot of rooms for improvement for the performance of Java bytecodes to be close to native object codes.

3. The results suggest that although there are potentials in the mobile code technologies in the future, a native (C/C++) implementation is favored under current technologies if multimedia data processing capabilities are need in the VM.

3.2 Experiment B: Profiling of the Java MPEG Video Player

Experiment B is designed to understand which parts of the MPEG video processing algorithm consume most of the time and which part of the algorithm can be extracted and implemented as common operations in the specialized virtual machine.

MPEG decoding process consist of four major steps: First, the input stream is read up to the complete Variable Length Code (VLC) and a Huffmann decoding process is called to convert it into integer a value. After all the VLCs belonging to a block are processed, the block data are inversely quantized. The Inverse Discrete Cosine Transform (IDCT) are then applied to the inverse-quantized block. The result is then added to the prediction data (reference pictures) to get a completed decoded frame. Figure 4 shows block diagram of the decoding process.

In order to understand which parts of the decoding process can be singled out and generalized to other applications, we ran a detail profiling of the Java MPEG video player using data set JP to determine the relative percentage of running time for the above four steps in the decoding process. The results are shown in table 5.

Module	Percentage of Running Time
I/O	29.89%
Huffmann decode	23.80%
Parse_Block	17.76%
IDCT	4.89%
Parse_macroblock	4.29%
Others	19.38%

Table 5: Running time distribution of MPEG video decoder components

Please note that the inverse quantization is not a separate module. The time spent in inverse quantization is included in the module Parse_Block.

The result shows that, to our surprise, IDCT does not occupy a very significant portion of the running time. It merely occupy less than 5 percent of the total running time. The Huffman decode module occupies almost 24% of the running time (excluding the I/O operations, which is categorized under I/O), therefore is a possible candidate for further optimization. The other modules which occupy top portions of running times are Parse_Block and Parse_Macroblock. These two combined occupy more than 22% of the total running time. Since the bodies of these two modules are not computing-intensive operations but mainly control logic for the decoding process, it is not easy to separate part of the code to optimize further.

We made some observations on the results:

- Excluding I/O operations, the control logic of MPEG decoding itself occupies 31.45% of the total running time. Comparing it to the Huffmann decoding which occupies 33.95% and IDCT which occupies 6.97% of total running time, the control logic of MPEG seems to be too complex. Since MPEG standard is very general and covers a wide range of configurations, it may be feasible to constraint the sets of configuration parameters to some pre-defined sets such as those in **H-261** to facilitate the processing when the frames are delivered through the networks.

- The results suggest that it may be desirable to implement the complete decoding process as a module instead of separating it into several modules to maximize the performance. Currently there are some commercial attempts to define the Java Media Frameworks [10] and Java Advanced Imaging API [10] in future Java standard. This line of thought is similar to our idea and the results from their work may provide some insights into this issue.

	Exp A		Exp B		Exp C	
OS	Linux	% of orig.	Solaris	% of orig.	MS Win 98	% of orig.
I frames	458.21	200%	2217.55	176%	198.07	323%
P frames	370.24	184%	1793.00	157%	166.27	270%
B frames	156.62	134%	802.39	125%	67.20	177%
Overall	235.26	160%	1168.44	143%	102.91	225%

Table 6: Running Time of Java/C hybrid implementation.

3.3 Experiment C: Java MPEG Video Player with Native Methods

To determine if we can gain any efficiency by implementing the most time-consuming parts of the decoder using more efficient languages such as C/C++, we implemented the Huffmann decoding module (which we identified in section 3.2 as the most time-consuming module in the decoding process) as a native method in C using the Java Native Interface (JNI) specifications. We repeat the same experiments as in section 3.1 on the new Java/C hybrid implementation. Table 6 show the results of our experiments.

The result show that instead of gaining efficiency through native method implementation, the hybrid implementation actually runs much slower than pure Java implementation. In some cases where the VM implementation is very efficient, the slow down is even more significant (323% slower). Similar results are also observed when the IDCT module is implemented as a native method. After further investigation, we contribute the slow down of the decoding to the following reason:

- The overhead of interacting between the native method and the Java methods are too high. When a native method needs to access a member variable, it has to perform a series of table lookup to find the class ID and field ID to access the correct data item. When a native method needs to call a non-native method, it also need to perform a series of table lookup to find the class ID and method ID. Although in our implementation we are very careful to only perform these lookups once during the class initialization, the overheads of calling the method or accessing the member variable through IDs are still very high.

- The overhead of setting up the native method call are too high. To validate this point, we conduct another experiment to determine the overhead in setting up native method calls. We construct two Java programs using the skeleton of the Huffmann decoding module in the experiments. One program is implemented as a pure Java program and the other are implemented as a native method. We keep the parameter list and the return value to be the same as those in

Type of call	Linux	Solaris	MS Windows 98
Java Method	61.3 ms	946.5 ms	28.00 ms
Native Method	1418.8 ms	3942.70 ms	340.00 ms
Native/Java	23.15	4.17	12.14

Table 7: Comparison of Java method and Native method calls

the Huffmann decoding module. The body of the methods in both program are empty so we can measure the overhead of setting up the call. We call the empty method 100000 times in both program and measure the time elapsed. The results are shown in table 7. The result clearly shows that the native call is very slow compared to the ordinary method calls. Under Solaris the native method calls are 4 time slower, which is the best case in our three test configurations. Under Linux it is 23.15 time slower, which suggests that there may be problems on the implementations of JDK under Linux.

From the above observation and discussion, it is clear that the current interface between the mobile code and the native calls are too complex and not efficient enough. If we want to implement specialized libraries methods, a simple and efficient way of interfacing the mobile codes and the specialized libraries methods is needed.

3.4 Experiment D: Object Code Size Statistics

Since packets in an active network environment carries active programs within the packet, the sizes of the active programs have very big influence on the feasibility of the architecture. For example, Smart Packets architecture [13] tries to limit the active program size within one Ethernet packet (1500 Bytes) by introducing a specialized assembly language. To get some estimates of how large a object code could be in complicated algorithms such as MPEG decoding, we tabulate the sizes of the object codes produced by the Java 2 compiler on the original Java MPEG decoder and the modifications we have made on Huffmann decoding and IDCT modules. The results are in table 8. Form the result, it is apparent that the size of almost any single functional module are bigger than the size of one Ethernet packet (1500 bytes). Some module such as Huffmann decoding is actually more than 10 times larger than a Ethernet packet. Therefore it is not feasible to put the active program written in Java bytecode within the active capsule for these kinds of complicated operations. To alleviate this problem, there are several alternatives that can be further explored. One approach is to employ bytecode compression techniques such as those described in [7, 5]. To determine the effect of compressions, we also conduct an experiment which compress the bytecodes of modules in table 8 using `gzip`. The results are shown in table 9. It is clear that even after the bytecodes are gzipped, the size is still very

Class Name	Element	Err	Huffmann	IDCT
Size	1697	283	16649	4136
Class Name	MPEG_Play	MPEG_scan	MPEG_video	dispatch
Size	6250	3490	14452	1032
Class Name	io_tool	motion_data	myFrame	semaphore
Size	3291	6327	426	538
Class Name	Huffmann (Native)	IDCT (Native)		
Size	729	511		

Table 8: Size of Java bytecodes on various modules (unit: byte)

Class Name	Element	Err	Huffmann	IDCT
Size	1075 (63%)	235 (83%)	6530 (39%)	2199 (53%)
Class Name	MPEG_Play	MPEG_scan	MPEG_video	dispatch
Size	3370 (55%)	1989 (57%)	7475 (52%)	679 (66%)
Class Name	io_tool	motion_data	myFrame	semaphore
Size	1895 (58%)	3038 (49%)	330 (77%)	389 (72%)
Class Name	Huffmann (Native)	IDCT (Native)		
Size	522 (72%)	391 (56%)		

Table 9: Size of Gzipped Java bytecodes on various modules (unit: byte)

large. According to [5], it is possible to compress the bytecodes to about 26% to 30% of the original size, therefore the results in table 9 may be further improved. However, bytecode compression need extra time to compress and decompress, therefore my not be feasible in a highly dynamic environments such as active routers. Another approach is to design code transport schemes to amortize the cost of code transporting. For example, ANTS [15] employ the on-demand code loading scheme using the MD5 cryptography hash as the signature of the protocol to minimize the overhead of setting up the protocol (specialized functions) dynamically. Router Plugins [8] tries to extend the functionalities of the router by *Plugins* - software modules that are dynamically load into kernel and configure on-demand at run time. These approaches provides some part of the solutions to the problem. However, ANTS treats a whole protocol as a unit and assigned a signature to it, which is too coarse-grain in our design. Router Plugins also is too coarse-grain in our case because it works on the whole packet.

In our design, we propose a new scheme called *Installable Operation Code* to solve the above problem. Simply speaking, the VM have a special operation to register a piece of code as a new opcode. When a piece of codes is registered as new opcode, the subsequent call can simply use the special function call `USR` (one byte) plus the code identifier (16 bytes for secure mode using MD5 or 16 bits for insecure mode using random number). Installable opcode is similar to the macro instructions in other programming languages (especially assembly languages). Our scheme is different from ordinary macro instructions in the following aspects: first, our design includes two versions of user-defined opcode to suit the security need for different applications. For applications with high security requirements, the applications can use MD5 opcode, but have to pay extra spaces (16 bytes) for the added security. For applications who don't need high security, a 16 bit opcode can be used to save packet spaces. Second, we allow the user-defined opcode to be applicable across several packets.

From the above result, we can also conclude that any operations which are supposed to be widely available among the execution environments (such as MPEG codec) should be built in to the virtual machines to avoid the high overhead of mobile codes.

4 Conclusions and Future Work

In this paper, we experimentally evaluated some tradeoffs in the design of specialized virtual machines for multimedia data in active networks. We try to provide some guidelines for choosing between alternatives based on the resource constraint. We also provide a new idea called *Installable Operation Code.* Although some part of the results may be obvious or not clearly favor any particular design, our result do give us some ideas of the performance for a particular choice. Some future work includes:

- More detail investigation of the overhead in native implementation.

- Performance evaluation of the *Installable Operation Code* scheme.
- Performance evaluation of other compressed video/audio processing algorithms.

References

[1] Active Network Working Group (K. L. Calvert, ed.). Architectural Framework for Active Networks Version 1.0. URL `http://www.dcs.uky.edu/~calvert/arch-1-0.ps`, July 1999.

[2] Elan Amir, Steven McCanne, and Randy H. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proceedings of the ACM SIGCOMM*, Vancouver, British Columbia, Canada, September 1998.

[3] Elan Amir, Steven McCanne, and Hui Zhang. An Application Level Video Gateway. In *ACM Multimedia'95, San Francisco*, 1995.

[4] Joerg Anders. Inline MPEG - 1 - player in Java. URL `http://rnvs.informatik.tu-chemnitz.de/~ja/MPEG/MPEG_Play.html`.

[5] Quetzalcoatl Bradley, R. Nigel Horspool, and Jan Vitek. JAZZ: An Efficient Compressed Format for Java Archive Files. In *Proceedings of CASCON'98*, pages 294–302, Toronto, Canada, November 1998.

[6] Kenneth L. Calvert, Samrat Bhattacharjee, Ellen Zegura, and James Sterbenz. Directions in Active Networks. *IEEE Communication Magazine*, October 1998.

[7] Lars Ræder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for embedded systems. Technical Report PI-1213, IRISA, December 1998.

[8] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of the ACM SIGCOMM*, Vancouver, British Columbia, Canada, September 1998.

[9] Osvaldo Pinali Doederlein. The Java Performance Report. URL `http://www.javalobby.org/features/jpr/`.

[10] Sun Microsystems. Java^{TM} Media APIs. URL `http://java.sun.com/products/java-media/`.

[11] John Neffenger. Java Benchmarks. URL `http://www.volano.com/benchmarks.html`.

[12] K. Patel, B. Smith, and L. Rowe. Performance of a Software MPEG Video Decoder. *In Proceedings of ACM Multimedia 93*, August 1993.

[13] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *Proceedings of the Second IEEE Conference on Open Architectures and Network Programming*, New York, N.Y. USA, March 1999.

[14] Sheng-Yih Wang and Bharat Bhargava. An Adaptable Network Architecture for Multimedia Traffic Management and Control. Technical Report CSD-99-048, Purdue University, Department of Computer Sciences, November 1999.

[15] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of IEEE OPENARCH'98, San Francisco, CA*, April 1998.