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

Sheng-Yih Wang and Bharat Bhargava

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

ABSTRACT

In active networks environments, the data packets can carry active programs to enable specialized processing on them. We quantify the effectiveness of general capsule programs v.s. specialized processing functions for multimedia data through four experiments. These experiments deal with the comparison of Java v.s. C implementation of a MPEG video decoder, the identification of the time-consuming modules in a Java MPEG video decoder, the effectiveness of combining Java methods and native methods, and the size of the bytecode for each module in a Java MPEG video decoder. We found that Java MPEG decoder can be 2.6 times to 10 times slower than an equivalent C implementation. We identified Huffmann decoding module as the most time-consuming module. We also found that the Native Method Interface (NMI) is complex and not efficient to be used in active routers and the size of the bytecodes for most of the modules is too big to fit into a single packet even after compression. We draw certain conclusions about the trade-off between the general programming model and the specialized functions provided by the router for the active capsules.

1. INTRODUCTION

Active Networks [1] and Active Services [2] are important research topics. 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 [2]. Since the emerging applications will contain a lot of multimedia data such as video and audio data, how to provide specialized multimedia data processing capabilities within these new service architectures is a challenging issue. Current active network design such as Smart Packets [8] or ANTS [10] 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 [9], 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.

We quantify the effectiveness of general capsule programs v.s. specialized processing functions for multimedia data through a series of experiments. We draw conclusions on the trade-off between the general programming model and the specialized functions provided by the router for the active capsules. We obtain certain quantitative data for those trade-offs. These data can be used to guide our design of the active network architecture. These data can be interpolated or extrapolated to predict the hardware requirements (CPU speed, memory speed/capacity, etc.) for emerging active network architectures.

PORTIONS OF THIS WORK WERE SUPPORTED BY A GRANT FROM NSF UNDER CCR-9901712, A GRANT FROM IBM, AND BY SPONSORS OF THE CENTER FOR EDUCATION AND RESEARCH IN INFORMATION ASSURANCE AND SECURITY (CERIAS).

2. BACKGROUND

In an active network architecture, the nodes can be active or non-active. Active nodes run a Node Operating System (NodeOS) and one or more Execution Environments (EEs) that 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) or very specific (such as Spanner [8] in the Smart Packet project, which provides specialized functions for network management tasks). To understand the design issues of the specialized virtual machine, the following questions are important.

(1) Are general script or virtual machines fast enough to implement complicated algorithm such as MPEG video codec? How fast can mobile code techniques achieve in comparison with the native code implementation?

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

(3) How big are 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 these questions are general and depend on many different factors such as the choice of virtual machine and video processing algorithm, we investigate one special case that can provide some insights into the real scenario. Specifically, we analyze the MPEG video decoding algorithm under C and Java implementations. Since MPEG video decoding is an essential part of compressed video processing such as transcoding, the data gathered is useful not only for MPEG algorithms but also for a broader range of multimedia applications. We are not trying to answer the general benchmark questions of Java virtual machines. Research on this topic is available in [6]. Our goal is to get an estimate of how mobile codes can perform in 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 (Jurassic Park, Tai Chi, and Lion King), which we digitized from various commercial video tapes. These video clips represent d-ifferent type of commercial-quality videos. The details of these video clips can be found in [9].

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: Comparison of Java and C implementation of MPEG video decoder

The purpose of this experiment is to determine if the Java virtual machines and bytecodes are fast enough to implement algorithm-s such as MPEG video decoding. It demonstrates the efficiency of currently available mobile code techniques in comparison with the native code implementation. We compare the decoding time of both Java and C implementation of the MPEG-I video decoder. For C implementation of the MPEG video decoder, we use the Berkeley *mpeg_play* [7] program. For Java implementation, we modify the decoder written by Professor Joerg Anders at Technische Universität Chemnitz, Germany [3]. We have compiled the C implementation under two configurations: (1) Linux which use 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 (results in table 1).

From the result, we made some observations:

(1) The Java decoder is, in general, 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 the result is not a surprise to us (we know that Java decoder will be slower than the C decoder.), it does provide an idea of the magnitude of slowdown when using Java on a real problem. Under Microsoft Windows 98, the slow down is not as big as other platforms but still slow (2.6 times slower comparing to data set TC under Linux/C)[1].

(2) Comparing experiments Set 3 and Set 4, we note that the JIT compiler provides substantial improvement in the execution time over the bytecode interpreter (by a factor of four). However, there is room for significant improvement for the performance of Java bytecodes to be close to native object codes.

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

[1] Since *mpeg_play* is an X-Windows based program, it is not easy to port to MS Windows 98. Therefore we didn't 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.

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 2: Experiment: Running time distribution of MPEG video decoder

3.2. Experiment: Profiling of the Java MPEG Video Decoder

In this experiment, we try to determine which parts of the MPEG video processing algorithm consume a major portion of the time and which parts 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 (VL-C) and a Huffmann decoding process is called to convert it into an integer value. After all the VLCs belonging to a block are processed, the block data are inversely quantized. The Inverse Discrete Cosine Transform (IDCT) is then applied to the inverse-quantized block. The result is then added to the prediction data (reference pictures) to get a completed decoded frame.

We ran a detail profiling of the Java MPEG video decoder using data set JP to determine the relative percentage of running time for the above four steps in the decoding process. (Results are shown in Table 2). 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 significant portion of the running time. It occupies less than 5% 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), and 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 these two modules are not computational-intensive operations but mainly control logic for the decoding process, it is not easy to separate parts of the code to optimize further.

Our observations on these results are as follows:

(1) 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 IDC-T 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 constrain 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.

(2) 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 APIs for multimedia data in future Java standard [5]. This line of effort is similar to our idea and the results from their work may provide additional insights.

	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
Jurassic Park	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
Tai Chi	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
Lion King	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 1: Average decoding time for three video clips

	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 3: Running Time of Java/C hybrid implementation

3.3. Experiment: Java MPEG Video Decoder with Native Methods

To determine if we can gain any efficiency by implementing the most time-consuming parts of the decoder using more 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. (Results are in Table 3).

The result shows 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 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:

(1) The overhead of interaction between the native method and the Java methods is 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 needs to perform a series of table lookup to find the class ID and method ID. Although in our implementation we are 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.

(2) The overhead of setting up the native method call is high.

Type of call	Linux	Solaris	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 4: Java method v.s. Native method calls

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 is implemented as a native method. We keep the parameter list and the return value to be the same as those in the Huffmann decoding module. The body of the methods in both programs are empty so we can measure the overhead of setting up the call. We call the empty method 100,000 times in both programs and measure the time elapsed (Results are in Table 4). The result clearly shows that the native call is 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 times 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 complex and not efficient. If we want to implement specialized method libraries, a simple and efficient way of interfacing the mobile codes and the specialized method is needed.

Class Name	Element	Err	Huffmann	IDCT	MPEG_Play
Orig. Size	1697	283	16649	4136	6250
Gzipped Size	1075 (63%)	235 (83%)	6530 (39%)	2199 (53%)	3370 (55%)
Class Name	MPEG_scan	MPEG_video	dispatch	io_tool	motion_data
Orig. Size	3490	14452	1032	3291	6327
Gzipped Size	1989 (57%)	7475 (52%)	679 (66%)	1895 (58%)	3038 (49%)
Class Name	myFrame	semaphore	Huffmann (Native)	IDCT (Native)	
Orig. Size	426	538	729	511	
Gzipped Size	330 (77%)	389 (72%)	522 (72%)	391 (56%)	

Table 5: Size of Java bytecodes on various modules (unit: byte)

3.4. Experiment: Object Code Size Statistics

Since packets in an active network environment carry active programs within the packets, the sizes of the active programs have significant influence on the feasibility of the architecture. For example, Smart Packets architecture [8] tries to limit the active program size within one Ethernet packet (1500 Bytes) by introducing a specialized assembly language. To get some estimates of the size of object codes, we tabulate the sizes of the bytecodes produced by the Java 2 compiler on the original Java implementation and the modified Huffmann and IDCT modules in Table 5. From the result, it is apparent that the size of almost any single functional module are much bigger (up to 10 times) than the size of one Ethernet packet (1500 bytes). Therefore it is not feasible to put the active program written in Java bytecode within the active capsule. 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 [4]. To determine the effect of compression, we compress the bytecodes of modules in Table 5 using `gzip`. Even after the bytecodes are gzipped, the sizes are still large. According to [4], it is possible to compress the bytecodes to about 26% to 30% of the original size, therefore the results may be further improved. However, bytecode compression needs extra time to compress and decompress. They may not be feasible in a highly dynamic environments such as active routers where most of the active program are only executed once and are discarded after execution. Another approach is to design code transport schemes to amortize the cost of code transporting. For example, ANTS [10] employs the on-demand code loading scheme to minimize the overhead of setting up the protocol (specialized functions) dynamically. However, ANTS and other approaches treat a whole protocol as a unit and, which has coarse-granularity.

From the above result, we can 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 tradeoffs in the design of specialized virtual machines for multimedia data in active networks. We provide guidelines for choosing between alternatives based on the resource constraints. Although some results may be obvious or not favor any particular design, our results do give us clear ideas of the expected performance for a particular choice. Some future work includes:

(1) More detail investigation of the overhead in native implementation.

(2) Performance evaluation of the bytecode compression schemes.

(3) 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] Joerg Anders. Inline MPEG-1 Player in Java. URL `http://rnvs.informatik.tu-chemnitz.de/~ja/MPEG/MPEG_Play.html`.

[4] 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.

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

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

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

[8] 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 City, New York, March 1999.

[9] 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.

[10] 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, California, April 1998.