Java Jitters - The Effects of Java on Jitter in a Continuous Media Server

Mark Claypool
Worcester Polytechnic Institute, claypool@wpi.edu

Jonathan Tanner
Worcester Polytechnic Institute, jtanner@cs.wpi.edu

Java Jitters - The Effects of Java on Jitter in a Continuous Media Server

Mark Claypool and Jonathan Tanner

Computer Science Department, Worcester Polytechnic Institute

{claypool,jtanner}@cs.wpi.edu

Introduction

The tremendous power and low price of today's computer systems have created the opportunity for exciting applications rich with audio and video. These new continuous media applications promise to enrich our lives by enhancing our stream-like interactions with the power and flexibility of computers. Java is equally promising with the potential to transform application development as we know it. The "write once, run anywhere" nature of Java bytecode continues to score major implementation wins, especially at large organizations whose need for cross-platform solutions overrides other factors. The Java Media APIs are designed to meet the increasing demand for continuous media, supporting audio, video, animations and telephony [JMA98]. The use of Java for continuous media applications is inevitable.

Before Java can be executed, it must first be compiled from source code into what is known as *bytecode*. There are several different ways of executing bytecode as native machine code: a Java Virtual Machine (JVM) is an interpreter that translates the bytecodes into machine code one by one, over and over again; a Just in Time (JIT) compiler translates some the bytecodes into machine code just before they are to be used and caches them in memory for reuse; and a static native compiler translates all the bytecode operations into native machine code, taking full advantage of traditional compiler optimizations.

Related work on Java performance has concentrated on the performance of traditional benchmarks such as Spec95 and the jBYTEmark in Java environments [HCJ+97, HG98]. CaffeineMark seeks to provide an indicator of Java Applet performance in a Java runtime environment [CM98]. Such research has shown that JIT and static native compilation can provide impressive performance improvements over purely interpreted Java. However, traditional benchmarks tend to model traditional application performance. Continuous media applications have very different performance requirements than traditional applications.

Although we often think of continuous media as a stream of data, computer systems handle continuous media in discrete events. An event may be receiving an update packet or displaying a rendered video frame on the screen. The quantity and timing of these events give us measures that affect application quality. There are three measures that determine quality for most continuous media applications [CR98]: *Latency*, the time it takes information to move from the server through the client to the user; *Jitter*, the variation in latency, can cause gaps in the playout of a stream such as in an audioconference, or a choppy appearance to a video display; and *Data Loss* which can take many forms such as reduced bits of color, pixel groups, smaller images, dropped frames and lossy compression.

Delay and loss are the primary concerns for traditional text-based applications. Jitter, however, is a concern unique to continuous media performance. In the absence of jitter, continuous media frames can be played as they are received, resulting in a smooth playout, depicted by Figure 1. However, in the presence of jitter, inter-frame times will vary, as depicted in Figure 2. In Figure 2, the third frame arrives late at time r*2*. In the case of audio speech, the listener would experience an annoying pause during this period. In the case of video, the viewer would see the frozen image of the most recently delivered frame.

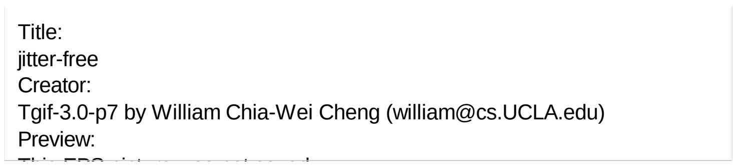

Title:
jitter-free
Creator:
Tgif-3.0-p7 by William Chia-Wei Cheng (william@cs.UCLA.edu)
Preview:

Figure 1. The above figure is a model of a jitter-free stream. Each s*i* is the time at which the server initiates the transmission of frame *i*. Each r*i* is the time at which the client plays frame i.

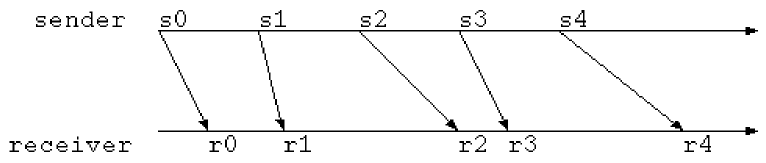


Figure 2. The above figure is a model of a stream with jitter. Each s*i* is the time at which the server initiates the transmission of frame *i*. Each r*i* is the time at which the client plays frame *i*.

In past research, we have empirically shown that an increase in processor load results in an increase in jitter [CR97]. Interpreted Java has the added processor load of the interpreter, making increased jitter seem likely. Moreover, object-oriented languages such as Java make heavy use of memory. Java removes the burden of memory management from the programmer through runtime garbage collection. This freedom comes at a performance price, however, as JVMs often spend 15 percent to 20 percent of their time on garbage collection [HG98]. Most significantly, a chart of the memory usage of a JVM shows a jagged sawtooth pattern (see Figure 3, from [HG98]), indicating that garbage collection is intermittent.

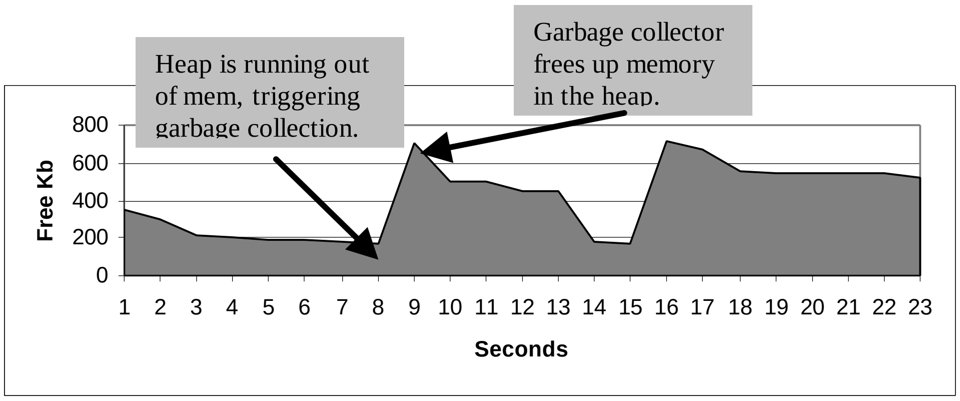


Figure 3. This jagged pattern is typical of memory availability when the garbage collector disposes of groups of objects.

We hypothesize continuous media applications suffer from increased jitter under most Java runtime environments because of periodic garbage collection and interpreter processor load. We further hypothesize compiled Java achieves nearly the same frame rate as C++ but suffers from jitter because of the overhead of garbage collection in the runtime environment.

Experiments

To evaluate Java performance and test our hypotheses, we designed a portable, platform-neutral server benchmark that simulated a continuous media server. Our server benchmark captures the fundamental components of a continuous media server:

- Capture – obtaining data from the codec or data file (in our benchmark, a data file).

- (De)Compress – formatting the data to a format suitable for the client (in our benchmark, incrementing a long integer variable in a tight loop, and then modifying the frame array values to represent compression or decompression).
- Send – transmitting the data over the network (in our benchmark, written to an output file).

Read, (De)Compress and Send are tunable parameters, enabling benchmarking of different continuous media formats. For example, a video server sending a MPEG video stream would require different read sizes, compression amounts and send sizes than an audio server sending an voice stream. In addition, altering the limits of each parameter enables the location of performance bottlenecks. The following table shows the steps carried out by our server benchmark compared to steps that would be carried out by an actual continuous media server:

Server	*Simulated Server*
1. Establish connection	1. Open output data file
2. Verify video feed	2. Open input data file
3. Capture frame data	3. Read frame data
4. Compress/decompress	4. Simulate decompression
5. Send frame data	5. Write frame data to output file
6. Sleep until next frame	6. Sleep until next frame

One of the concerns with simulations is that they may abstract away too many details that affect real application performance. In order to address this concern, in our previous work, we conducted experiments that verified that the performance of similar simulations comes within 10% of the performance of real applications built with the same components [CR96].

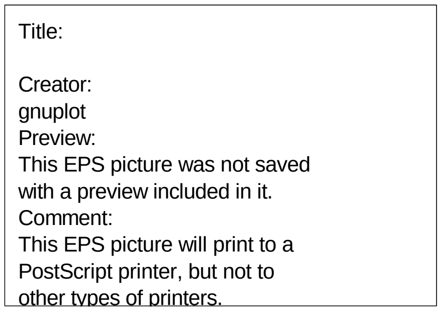
We implemented our server in Java and ran it under two Java runtime environments: interpreted and static native compiled. For the interpreted Java, we used the JVM from Sun's JDK 1.0.2, and for the static native compiled Java, we used Toba v1.0, a freeware static native compiler [PH98]. In order to obtain a performance baseline for the Java results, we implemented our server in C++ and ran it under the same hardware configuration as the Java server. The hardware configuration used for all experiments was a dedicated PC with an Intel Pentium 166 MHz, 32 MB RAM, 512 KB cache, EIDE hard disk running Linux 2.0.30. For each experiment run, we captured 50 MB of continuous media data. We recorded the time between successive frames which we used to derive the inter-frame times.

In order to locate bottlenecks in the Java runtime environments across a variety of different types of continuous media servers, we tuned the server components, varying the:

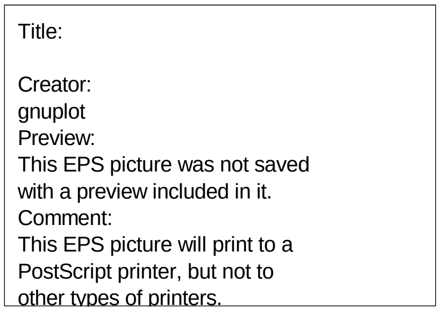
- Frame Rate from three frames per second up to thirty frames per second. Previous research has shown four three frames per second, as being the minimum frame rate needed for remote tasks. Thirty frames per second is equivalent to full-motion video.
- Read Size from 4k to 76k. 4k was chosen as the average frame size in an MPEG format video file while seventy six kilobytes was the size of an uncompressed frame of 320x240 at 256 colors.
- Compression Rate over minimal, partial and maximal levels of compression.
- Send Size from 4k to 76k, to match the read size.

Results and Analysis

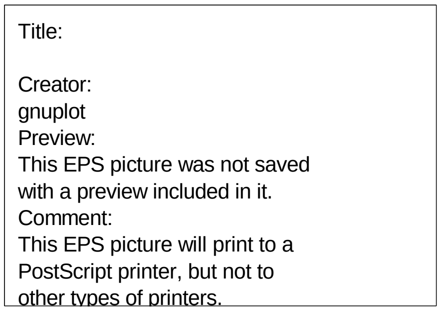
The effects of Java runtime environments on jitter were striking. Figure 4 shows a visual comparison of the jitter in the JVM, static native and C++ streams, for a 5 frame-per-second stream with average compression. Visually, the JVM and Toba environments had the same amount of jitter while C++ by far had the least. We found similar amounts of jitter for all read sizes, send sizes and compression levels. Surprisingly, jitter increased only slightly under higher levels compression, even with the JVM.


Title:

Creator:
gnuplot
Preview:
This EPS picture was not saved
with a preview included in it.
Comment:
This EPS picture will print to a
PostScript printer, but not to
other types of printers.


Title:

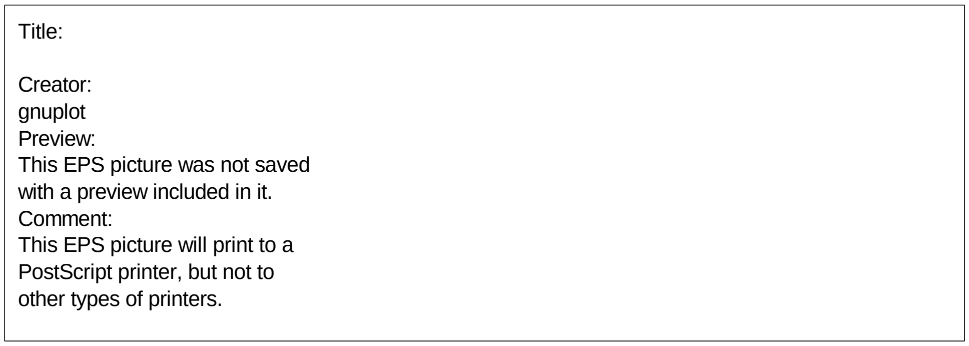
Creator:
gnuplot
Preview:
This EPS picture was not saved
with a preview included in it.
Comment:
This EPS picture will print to a
PostScript printer, but not to
other types of printers.


Title:

Creator:
gnuplot
Preview:
This EPS picture was not saved
with a preview included in it.
Comment:
This EPS picture will print to a
PostScript printer, but not to
other types of printers.

Figure 4. Observable jitter. Three graphs of server jitter are depicted. From the left, JVM, Toba and C++. The horizontal axes are seconds. The vertical axes are interframe times in milliseconds. The frame rate is 5 frames per second with average compression.

Java runtime environments had an equally noticably reduced framerate. Figure 5 depicts the results for a MPEG-type server with minimal compression. If the runtime environment were able to keep up with the expected frame rate the curve would fall exactly on the "expected" curve in the graph, as was almost the case for the C++ runtime system. However, both the JVM and Toba were unable to achieve more than 10 frames per second, as is evident by their curves departing the "expected" curve. We found similar results for all read and send sizes. Higher levels of compression, however, resulted in a reduced frame rate for Java and Toba, while C++ was able to achieve full-motion frame rates for all levels of compression.


Title:

Creator:
gnuplot
Preview:
This EPS picture was not saved
with a preview included in it.
Comment:
This EPS picture will print to a
PostScript printer, but not to
other types of printers.

Figure 5. Framerate. This graph depicts the expected plus actual frame rate for server runs with an increasing frame rate. The horizontal axis is the framerate required by the client. The vertical axis is the time between frames. The expected time is graphed as a baseline. Runs for JVM, Toba and C++ are shown for framerates from 3-25 with minimal compression.

Conclusions

We find Java can achieve only 1/3 the framerate of full-motion video (30 frames/second). However, if the overhead of garbage collection is improved, we predict interpreted Java could achieve 2/3 the framerate of full-motion video. Static native compiled Java suffers from nearly as much jitter as interpreted Java. In all cases, C++ is still vastly superior in amount of jitter and maximum framerate to both interpreted and compiled Java.

In summary, the contributions of this work are:

- To the best of our knowledge, we are the first to provide experiment-based Java runtime performance for continuous media applications.
- In addition, we provide a continuous media server benchmark that is tunable to different media formats. Our server allows us to benchmark Java runtime systems for different continuous media servers and compare performance to C++ runtime systems.

Will Java always be noticeably slower than C++? While still a matter of debate, many experts agree that Java will achieve at least 60-70% the speed of native performance. However, as evidenced by the jitter results in this paper, typical speed improvements do not always result in an equal improvement to continuous media performance. In particular, the *timing constraints* imposed by continuous media applications must be accomodated as well, if continuous media jitter is to be improved.

Future Work

As mentioned in the introduction, a particularly promising means of improving Java runtime performance is through JIT compilation. We have obtained some preliminary results using Kaffe, a freeware JIT compiler [Wi98] that suggests that Kaffe, like Toba, performs similarly the Sun JVM under our continuous media benchmark. Future work is needed to determine these results under reproducable conditions.

Cross-platform compatibility is one of the most exciting promises offered by Java. This will most likely bear out in numerous Java continuous media clients to connect to the continuous media servers. We are developing a benchmark for continuous media clients that will allow Java runtime performance comparisons similar to those that we have demonstrated here for continuous media servers.

References

[CM98] CaffeineMark Java Benchmark: The industry standard Java benchmark, Pendragon Software. *Internet site*: `http://www.webfayre.com/cm.html`

[CR96] Mark Claypool and John Riedl. A Quality Planning Model for Distributed Multimedia in the Virtual Cockpit. In *Proceedings of ACM Multimedia*, pages 253-264, November 1996.

[CR97] Mark Claypool, Joe Habermann and John Riedl. The Effects of High-Performance Processors, Real-Time Priorities and High-Speed Networks on Jitter in a Multimedia Stream. *Technical Report TR 97-023*, Computer Science Department, University of Minnesota, May 1997.

[CR98] M. Claypool and J. Riedl, End-to-End Quality in Multimedia Applications, Chapter 40 in *Handbook on Multimedia Computing*, CRC Press, Fall 1998.

[HCJ+97] C. Hsieh, M. Conte, T. Johnson, J. Gyllenhaal and W. Hwu, Optimizing NET Compilers for Improved Java Performance, *IEEE Computer*, June 1997.

[HG98] T. Halfhill and A. Gallant, How to Soup Up Java, *Byte Magazine*, Vol. 23, No. 5, May 1998.

[JMA98] Java Media APIs, Sun Microsystems, 1998. *Internet site*: `http://java.sun.com/products/java-media`

[PHT+97] T. Proebsting, Gregg Townsend, Patrick Bridges, J. Hartman, Tim Newsham, Scott A. Watterson, Toba: Java for Applications, *Technical Report TR97-01, University of Arizona*, 1998. *Internet site*: `http://www.cs.arizona.edu/sumatra/toba`

[Wi98] T. Wilkinson, Kaffe - A Free Virtual Machine to run Java Code, 1998. *Internet site*: `http://www.kaffe.org/`