ICMC 2007 - Proceedings of the International Computer Music Conference, "Immersed Music". Vol. II, pp. 99 - 102.

REAL-TIME, LOW LATENCY AUDIO PROCESSING IN JAVA

Nicolas Juillerat, Stefan Müller Arisona, Simon Schubiger-Banz
ETH Zurich
Computer Systems Institute

ABSTRACT

This paper discusses the implementation of real-time and low latency audio processing in Java. Despite the fact that Java SE is widespread and has a large programmer base, it is clearly neither targeted at real-time, nor at low-latency applications. As such, doing good audio processing with this language is a challenging task and various issues have to be taken into account: these include limitations or properties of the audio drivers, the kernel and operating system, the audio API, the Java virtual machine and the garbage collector. We present a concrete Java audio processing framework called Decklight 4 that takes up the challenge. We show that, despite of many constraints, it is possible to get latencies as low as a few milliseconds on a standard personal computer using Java. We present the various elements of our implementation that allow such a result to be achieved, and we validate them through experimental measurements.

1. INTRODUCTION

The quality of a real-time audio processing tool is tightly related to its latency. Real-time voice transformations for instance should provide a sufficiently low latency so that the speaker or singer is not disturbed by the delay. While delays of less than 20 milliseconds are nearly unnoticeable with voice and most melodic instruments, delays as low as 4 milliseconds can be noticed with drum-like sounds [3].

Because of these strong requirements, real-time audio processing tools have been mostly implemented in C or C++. We pretend that a Java implementation achieving a few milliseconds of latency is possible, despite of various limitations that are commonly stated: the absence of real-time guarantees, or simply the slower speed compared to C. In this paper, we present the key issues that such an implementation has to deal with.

We choose the Java language because of its convienence, wider developer base and portability.

Unlike previous research on this topic with Java, we do not want to present the architecture, design and usage of such a tool, but rather the various aspects that affect the latency and how to cope with them in the implementation.

The rest of this paper is structured as follows: in section 2, we present all the aspects of a Java real-time audio processing tool that can affect the latency and we propose ways of minimizing it. Then we present a concrete implementation based on our theories in section 3 and give practical results. We compare our work to related research in section 4 and suggest future work in section 5.

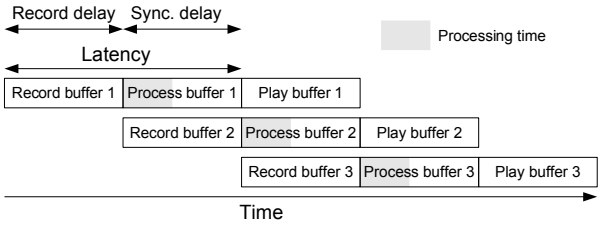

Figure 1. Audio processing pipeline

2. JAVA AUDIO PROCESSING

In this section, we discuss the elements of a digital audio processing tool, with focus on issues specific to the use of the Java programming language and latency. We start with the hardware and driver access in sections 2.1 and 2.2. We continue with the operating system and virtual machine issues in sections 2.3 and 2.4. Finally, we discuss issues related to the processing tool itself.

2.1. The Audio Pipeline

Ideally, an audio signal processor could record a sample, process it, and play the result, providing nearly zero latency. With the exception of some professional devices, this is unfortunately not achievable in practice. Record and playback are performed by sound cards on blocks of samples, or *buffers*. While the size of a buffer can usually be configured, the sole decomposition of the signal into buffers introduces a minimal latency to the system.

Figure 1 shows the typical pipeline of a real-time processing application. This is an ideal pipeline, in which we assume that the application has direct control over the underlying audio hardware. By using buffers, at least two delays are introduced that both contribute to the total latency of the system:

- The record delay. Samples cannot be processed until at least one buffer has been filled.
- The synchronization delay. The processing, even if it is very fast, takes some time. As a result, the playback usually cannot start until the next record buffer is ready.

Thus, the latency is given by at least twice the buffer size. Depending on the audio driver and API, additional delays can take place. These can be due to any internal mixing, resampling or processing that takes place before the samples are sent to the sound card.

The allowed sizes for the buffers greatly vary from one operating system to another, and from one sound card to another. On Linux architectures, we could set it as small as 16 samples with most sound cards when working with the Advanced Linux Sound Architecture (ALSA) [8]. On other platforms, it was sometimes not possible to go below 512 samples though (e.g., Windows XP with DirectSound and an old soundcard).

Finally, some additional delays can be introduced by the internal wirings of the sound card itself. These delays are expected to be relatively small.

2.2. Choosing the Audio API

We have depicted the ideal pipeline of an audio processing tool in the last section. To get as close to it as possible in practice, it is important to have a low-level audio API that is close to the hardware, and allows us to control the exact buffer sizes.

Preliminary tests with JavaSound [10], the standard sound API of the Java platform, revealed that it internally uses various intermediate mixing buffers. Because the application has nearly no control over them, achieving low latencies was not possible. The best we could achieve was as long as about 150 milliseconds, and the results greatly varied from one architecture to another.

A better solution is to use the ALSA API directly. ALSA API calls can be wrapped using the Java Native Interface (JNI) with a few lines of C code. This approach would limit us to the Linux platform only. Thus our implementation uses RtAudio [6], which redirects the calls nearly directly to ALSA, but also allows us to use DirectSound on Windows or CoreAudio on MacOS.

Another problem is to get a precise clock for timing. Our tests showed that none of the clocks provided by Java or by the Linux kernel was accurate enough for audio processing. We ended up using the number of captured audio samples as a clock, which gave the best results in practice.

2.3. Dealing with the HotSpot Compiler

When Java code is executed, it is not immediately compiled. Instead, when a loop is encountered for the first time, a few iterations are executed in interpreted mode, and then only, the code is compiled. This is the HotSpot feature of the Java Virtual Machine (JVM), which has been introduced since the release 1.4.

While the speed of compiled Java code is fast enough for real-time computations [4], the first few iterations that are executed in interpreted mode can be too slow. As a result, the computation time can take much more than the duration of one audio buffer in the first iteration. In other words, the synchronization delay discussed in section 2.1 may span many buffers. With most audio drivers, play buffers are queued, and the delay will never be reduced in the next iterations. As a result, the latency will stall with a large duration forever.

Because the processing time of the first iteration is so important for the latency, we have to play some tricks in order to get acceptable results with the HotSpot JVM. The simplest one is to run the whole processing in a "disconnected" way for a few iterations, that is, without actually recording or playing samples. Actual recording and playback are only started after a small period, when all the Java code has been compiled and is thus running at an optimal speed. The amount of time that is required for the JVM to compile everything varies from one version to another, and also depends on the amount of code to execute. We got delays ranging from 0.1 to 2 seconds in practice.

Another way of avoiding the latency induced by the first iterations is to force the synchronization delay discussed in section 2.1 to one buffer. This is possible with ALSA for instance, by setting the number of buffers to two. Samples are just dropped from the output if an iteration takes to much time, but the latency is not affected in the next iterations. Because samples are expected to be heavily dropped in the first few iterations anyway, we only use this feature in addition to the previous trick. Using this feature also ensures that an occasional long computing time does not affect the latency but just drops samples.

Remaining variations between the processing duration of consecutive iterations are unavoidable. But they can be drastically reduced by using threads with real-time priorities whenever supported by the operating system. Switching a Java thread to real-time was implemented by invoking the corresponding OS routines through a JNI wrapper. Using real-time priority allowed us to use audio buffers of 16 samples without being affected by intensive concurrent processing. Without it, at least 64 samples were necessary to get good results with no other threads running, and much larger buffers with other CPU-intensive threads running concurrently.

2.4. Garbage Collection and Priority Inversion

A problem that is related to the HotSpot compilation is jitter: it does not suffice that the processing time takes less than the duration of a buffer on average: if a single iteration takes more than one buffer, the playback stream will either be delayed or samples will be dropped.

A component of the Java virtual machine that can introduce large jitter at any time is the garbage collector. More precisely, the following features of the garbage collector may prevent the creation of a real-time application:

- In order to do its job, the garbage collector can occasionally block all threads of the Java application that are trying to allocate new objects, regardless of their priorities.

- The amount of time during which the threads are blocked is not guaranteed.

In other words, the garbage collector can introduce *any* amount of jitter, and makes Java inherently unsuitable for real-time applications. In practice though, we found that the duration of garbage collections with the latest Sun JVM was never above one millisecond. As a general rule of thumb, it is always good to avoid garbage in the first place but our experience showed that even regular memory allocations (few objects per cycle) during processing will not affect the overall performance. The IBM JVM [9] on the other hand typically took about 8 milliseconds per garbage collection with our application.

There is an additional point that is worth noting: we suggested in the previous section to use real-time priority threads. Unfortunately, the garbage collector runs in a separate thread over which we have no control. This thread does not run in real-time, but as a low priority thread because garbage collection is supposed to be a background task. Thus, when the garbage collector has to block the other running threads, the entire application is blocked, waiting on a low-priority thread that can be preempted at any time by other running applications. This phenomenon is known as priority inversion and can seriously affect the jitter. Fortunately, this problem is completely resolved by operating systems that implement priority inheritance: on such an operating system, if a low priority thread blocks a higher priority thread, the priority of the low priority thread is temporarily raised to that of the blocked thread. The latest Linux kernels implements priority inheritance, making jitter problems related to garbage collection nearly unnoticeable.

2.5. Inherent Audio Processing Latencies

Finally, there are some audio effects that have some inherent latency that adds up to the total latency of the system. Examples are all effects working in the frequency domain: it is necessary to work with buffers whose sizes must be large enough to capture the lowest frequencies we want to process. A pitch shifter for instance typically requires buffers of at least 10 milliseconds to perform acceptable results, and even more to perform good results. We refer to the buffers discussed in section 2.1 as the *audio buffers*, and to the buffers required for a particular audio effect as the *effect buffers*.

Setting the audio buffers size to the size of the effect buffers is a possible, but not optimal solution: both the record delay and the synchronization delay discussed in section 2.1 are proportional to this size. A better solution is to use a fraction of the effect buffer size for the audio buffer size: multiple captured audio buffers are joined together to form a single effect buffer, and each effect buffer is then split back into multiple audio buffers after the transformation to feed the audio playback device.

The advantage of this scheme is that the effect buffer size only affects the record delay. The synchronization delay still depends on the audio buffer size, which is a fraction of the effect buffer size only. There is a small issue though: the processing of an effect buffer must be fast enough to fit within the duration of an audio buffer. We have to ensure that the first part of the result, corresponding to an audio buffer, can be played in time. This can be a problem with CPU intensive effects if the audio buffer size is much smaller than the effect buffer size.

3. CURRENT STATE

In this section, we discuss the current state of our framework and give preliminary results on practical tests.

3.1. Decklight 4

Decklight 4 is our audio processing framework written in Java on Linux. It implements all the ideas presented in this paper. It uses RtAudio for audio I/O and sets real-time priority threads for the audio processing. Various audio effects are implemented. They range from simple time-domain effects such as filters and echoes, to complex Fourier-domain effects such as pitch shifting. The only part that is written in C is the access to RtAudio and to the Linux threading API. In both cases the C code is a straightforward mapping between the actual API and Java methods and is thus portable to all platforms where RtAudio is available (notably Windows and Mac OS X). The rest of the application, including all audio effects and the fast Fourier transform are written entirely in Java.

The actual architecture of the framework is based on our previous projects Decklight 2 and 3 [2] and is thus not described here again. The only notable difference in Decklight 4 is the introduction of the Java language, and the focus on audio processing only.

3.2. Results

In order to validate our theory, we implemented two audio effects on top of Decklight 4, and we compared the theoretical latencies against experimental measurements. To make the measurements, we sent the left output of a mono audio signal through our application, and we mixed the result with the right channel that was directly wired. The result was recorded in an audio editor on another machine, and the latency was computed using the delay between the left and right channels.

The first audio effect was a ten-band equalizer using a cascade of Butterworth band-pass filters. The latency inherent to this effect itself is negligible. The audio buffers were set to a size of 16 samples. The second effect was a pitch shifter working in the frequency domain. It uses overlapped windows of 512 samples at a sample rate of 48000Hz. Its inherent latency is thus about 10.67 milliseconds. The audio buffer size was set to 32 samples.

The application was running with the Sun JVM version 6 on a Linux machine. The sound hardware was a Realtek SoundBlaster card. The audio driver was accessed by RtAudio through JNI.

Table 1 shows the theoretical and measured latencies with these two effects. This table reveals that there is only a small difference between the theoretical and measured latencies. These differences correspond to the duration

Effect	Audio buffer	Theoretical latency	Measured latency
Equalizer	0.33 ms	0.67 ms	1.5 ms
Pitch shifter	0.67 ms	11.33 ms	12.49 ms

Table 1. Theoretical and experimental latencies

of one audio buffer plus about 0.5 millisecond. This can be due to various causes, such as internal mixing buffers used by the driver, or internal post-processings performed by the soundcard. More precise specifications would be required to get an accurate answer.

4. RELATED WORK

There are many other frameworks for low-latency audio processing written in C or C++. But very few researchers have investigated the Java language. A notable exception is JASS [7]. Its implementation has similarities with ours, especially regarding the audio hardware access, but it does not use real-time threads. It also does not directly allow the audio buffer size from being smaller than the effect buffer size, for effects requiring buffers. As a result, our framework typically performs better on similar architectures. The authors also do not give any information on what aspects are affecting the latency and how to cope with them, apart from the buffer sizes.

More recently, Java bindings to the Linux "Jack" audio system have been implemented [1], yielding low measured latencies as well. The authors focus on the architecture and not on low latency aspects though. On the other hand, our JVM and garbage collector related findings are applicable to other media types as well (see [5] for an overview of our Soundium/Decklight multimedia platform).

It is also worth noting that the latest version of JavaSound [10], as well as the third-party implementation Tritonus [12], have removed most of the intermediate buffers between the application and the underlying API, such as ALSA. But our tests revealed that the resulting latency is still above 20 milliseconds.

5. FUTURE WORK

The main obstacle to real-time processing in Java is the garbage collector. Although the version of the Sun JVM actually performed well in practice, it is not *guaranteed* to always perform well. The version of the IBM JVM for instance adds about 20 times more jitter but is still fully Java compliant. A new real-time Java implementation (JSR-001), including a real-time garbage collector, has just been released by Sun. We plan to port our framework to it.

Other future working directions are towards the reduction of the latency inherent to some audio effects, which is a problem that highly depends on the nature of the effects themselves.

We plan to process MIDI events in real-time as well in addition to raw audio streams. We are also working on a full-featured, GUI application using our framework.

6. CONCLUSION

We have investigated the problem of implementing a real-time, low latency audio processing tool with the Java language. We saw that many aspects are affecting the overall latency of the system, some of which are specific to Java. We proposed various ways of coping with them and we implemented our ideas into a concrete framework. While we could achieve excellent results (within a few milliseconds) on a standard personal computer, we also discovered two potential limitations: the first one is that the results can vary significantly with different hardware, driver, operating system or Java virtual machine. The second one is that although we could achieve very low latencies, these results are not guaranteed. We expect that the development of real-time Java implementations with real-time garbage collectors will give more predictability and guaranteed results in the near future.

7. REFERENCES

[1] Jens Gulden, "JJack - Using the JACK Audio Connection Kit with Java", *Linux Audio Conference*, Berlin, Germany, 2007.

[2] S. Müller Arisona, S. Schubiger-Banz, M. Specht, "A Real-Time Multimedia Composition Layer", *Proceedings of AMCMM*, ACM Multimedia, Santa Barbara, 2006.

[3] N. Lago, "The Quest for Low Latency" *Proc. of the International Computer Music Conference*, Miami, Florida, 2004.

[4] J.P. Lewis and U. Neumann, "Performance of Java versus C++", Computer Graphics and Immersive Technology Lab, University of Southern California, 2004.

[5] S. Schubiger-Banz, S. Müller Arisona, "Soundium2: An Interactive Multimedia Playground", *Proc. of the International Computer Music Conference*, San Francisco, 2003.

[6] Gary P. Scavone, "RtAudio: A Cross-Platform C++ Class for Realtime Audio Input/Output", *Proc. of the International Computer Music Conference*, Göteborg, Sweden, 2002.

[7] Kees van den Doel, Dinesh K. Pai, "JASS: A Java Audio Synthesis System For Programmers", *Proc. of the International Conference on Auditory Display*, Espoo, Finland, 2001.

[8] "Advanced Linux Sound Architecture", http://alsa-project.org

[9] "IBM Java Standard Edition", http://www.ibm.com/java

[10] "JavaSound API Programmer's Guide", http://java.sun.com/j2se/1.5.0/docs/guide/sound

[11] "Java SE Real-Time", http://java.sun.com/javase/technologies/realtime.jsp

[12] "Tritonus: Open Source Java Sound", http://tritonus.org