REAL-TIME MUSIC SYNTHESIS IN JAVA USING THE METRONOME GARBAGE COLLECTOR

Joshua Auerbach — IBM Research

David F. Bacon — IBM Research

Florian Bömers — Bome Software

Perry Cheng — IBM Research

ABSTRACT

Automatic memory management via garbage collection is the key to the safety, portability, and high productivity of modern programming languages like Java. However, until now no truly real-time garbage collector has existed for Java. As a result, the extreme real-time requirements of interactive music synthesis and processing have made it impossible to build such systems in Java.

We have developed a hard real-time garbage collector, called *Metronome*, around which IBM has built a production real-time Java virtual machine running on a real-time variant of Redhat Enterprise Linux. In this paper we demonstrate the real-time capabilities of our virtual machine with a MIDI music synthesizer that we built entirely in standard Java using the full object-oriented feaures of the language.

We show that even with the addition of another thread allocating 8 MB/second of data, our garbage collector is able to sustain error-free 44 KHz music synthesis down to buffer sizes of 1ms, achieving keyboard-to-speaker latencies of 5.0±0.75ms, comparable to a Kurzweil K2000R synthesizer (3.9±1ms) and suitable for high-fidelity interactive performance.

1. INTRODUCTION

Interactive music processing is one of the most demanding of all real-time applications, demanding worst-case latencies of 5-10ms, with only 1-4ms of jitter [5, 18]. In comparison, other paradigmatic hard real-time systems have much less stringent requirements: 20-40ms for helicopter flight control, 10-50ms for financial arbitrage, and 20-40ms for telecommunications switches.

As a result, the programming of such systems has required sacrificing many of the software engineering advantages enjoyed by programmers of high-level languages like Java: security, portability, ease of debugging, freedom from memory errors, and a large body of portable standard libraries.

The primary reason for this has been Java's use of garbage collection, which has introduced non-deterministic interruptions of tens or hundreds of milliseconds.

But garbage collection is the key to many of Java's most important advantages: freedom from memory errors reduces the number of run-time exceptions and makes the remaining ones much easier to find. It also prevents many kinds of security attacks. Furthermore, automatic memory reclamation makes programs simpler because the application does not need to be concerned about which functions have responsibility for allocating and de-allocating memory.

Over the past several years at IBM Research, we have developed a hard real-time garbage collector called *Metronome*, first as a research prototype [2] and then as the central component of a new real-time Java virtual machine product [16]. The technology has been adopted for time- and safety-critical systems by companies in a number of industries, including its use by Raytheon for the development of the software for the next-generation Navy destroyer.

The real-time Java virtual machine runs on top of a customized Linux kernel, co-developed by IBM and the open-source community. These modifications to Linux are making their way into the mainstream, with most of them being incorporated into the real-time edition of Redhat Enterprise Linux 5 (RHEL5 RT).

The combination of these technologies raises the possibility of programming hard real-time systems on commodity hardware using a widely-distributed open-source operating system and a highly portable, high-level object-oriented language.

In the work described in this paper, we set out to systematically investigate the capabilities and limits of our technology in the domain of interactive music synthesis. To this end we built a music synthesizer from scratch, entirely in Java. We made extensive use of Java's object-oriented features, including frequent dynamic allocation of objects. This allowed us to significantly simplify and yet also generalize the system, relative to our previous experience implementing synthesizers in C and C++.

We make no claims for the innovativeness of our synthesizer from a musical perspective; it is a wavetable synthesizer based on the SoundFont 2 standard. However, it is the manner of its construction and the achievable performance results that are radical.

After describing the real-time virtual machine and garbage collector, and the design of the synthesizer itself, we evaluate the system from a number of angles. First we run it with successively smaller buffer sizes and investigate the limits of the system in terms of the number of samples it must buffer in order to maintain error-free performance. We also compare the real-time virtual machine to other virtual machines with different approaches to garbage col-

lection, and show how they differ.

We then measure the externally observed end-to-end latencies of the system under different conditions, break out the latency/jitter contributions of the MIDI and audio processing portions of the system, and compare against an identically measured Kurzweil K2000R synthesizer.

2. THE REAL-TIME JAVA VIRTUAL MACHINE

Java is a highly dynamic language, which gives it much of its power. However, this dynamicity has typically come at a significant cost in latency and jitter. The primary language features contributing to these problems, in order of severity, are:

Garbage Collection In order to free memory, the collector has to scan some, or at times all, of the objects in the system. This leads to unpredictable pauses up to several hundred milliseconds.

Just-in-time (JIT) Compilation To achieve high performance with a highly dynamic language, a just-in-time compiler is used which periodically compiles "hot" methods, and sometimes recompiles them as new classes are loaded into the system. This results in non-deterministic pauses of up to tens of milliseconds.

Dynamic Class Loading. Classes are not loaded or initialized until they are referenced for the first time by running code. This leads to pauses of several milliseconds to load the classes themselves, plus however much time is required by the class initialization functions.

Multi-threading. Java provides threads as the fundamental construct for concurrent programming, but does not provide priority-based or any other real-time scheduling. This can lead to unpredictable delays by both competing Java threads and other operating system threads.

The IBM WebSphere Real Time Java virtual machine or RTJVM [1] provides solutions for all of these problems. We will concentrate on the real-time garbage collector in the next section.

JIT compilation is avoided with an ahead-of-time (AOT) compiler that takes a Java application packaged as a `jar` file, and produces a new `jar` file which contains all of the original (portable) Java class files, but also includes compiled machine code for each one [15]. When run with the RTJVM on the same machine architecture, the compiled code is used; a standard JVM will ignore the bundled machine code and simply treat it like a normal Java application.

AOT compilation is complicated by several aspects of the Java language. First of all, class references are resolved as the program executes, and there is no guarantee that details of a class will be the same at runtime as they were at compile time. Thus, an AOT compiler cannot use information about fields, methods, or classes when generating code, as a JIT compiler can. Furthermore, AOT compilation must assume a particular target processor and so the native code performance benefits only apply when the application executes on that particular processor. For these reasons, code compiled ahead of time will frequently perform less efficiently than code compiled by a JIT. However, such code performs *predictably*, which is what is needed for real-time applications. The shortfall in absolute performance is generally tolerable given adequate resources.

Unpredictable pauses due to class loading are avoided by preloading and preinitializing the classes that will be used in an application. In general, there are too many classes in the classpath to consider loading them all, and (in the presence of reflection and JNI) there is no fully automated way of finding the subset of classes that will actually be loaded. Hence, RTJVM does not build in any automated algorithm. Instead, we supply a tool (RaTCAT) via IBM's AlphaWorks web site. With programmer assistance, the tool identifies the classes that must be preloaded and provides a runtime library to perform the preloading. The work reported in this paper did not actually use RaTCAT but relied on more manual techniques.

The problem of assigning priority levels and policies to Java threads is addressed in the Real Time Specification for Java (RTSJ) standard [4], which defines a set of classes precisely for that purpose. The RTJVM fully implements the RTSJ standard, with the result that FIFO scheduling with at least 28 priority levels is available to Java code, as long as the underlying operating system supports this style of scheduling.

2.1. Real-time Linux

It has been noted that the operating system is often the weak link when attempting to implement music software [5]. In building the RTJVM, it was understood that execution on standard Linux would be imperfect. Consequently, IBM (in conjunction with the open source community) defined a Linux variant suitable for real time use. This began with the pre-existing `PREEMPT_RT` patch that is needed to make FIFO scheduling work in a completely preemptive fashion.

Several changes were made to locking algorithms in the kernel. Spinlocks were replaced with mutexes. Fast userspace mutexes were adopted. Priority inheritence was enabled for all mutexes to ensure correct behavior when high priority threads compete with lower priority threads for mutex ownership. Scheduling was substantially redesigned to achieve correct scheduling on multiple CPUs while avoiding the twin pitfalls of too much synchronization (single queue) or insufficient coordination (separate queues per CPU). The goal of having the N CPUs running the N highest priority tasks at all times was largely achieved. Finally, a new approach to high resolution timing was adopted, enabling timing accuracy in the microsecond or even nanosecond range.

3. METRONOME GARBAGE COLLECTION

Metronome [2] is a garbage collector designed to support hard real-time, with highly predictable latencies down to the millisecond level. It is a key part of the IBM WebSphere Real Time [16] product. Its technology differs from earlier approaches to "real-time" collection in several fundamental ways.

First, Metronome's approach to collection allows virtually all of the collection work to be done asynchronously by the collector. This is in contrast to previous approaches, which often required the application threads to perform work (like copying objects or updating pointers) on behalf of the collector, causing uneven and unpredictable performance of the application threads.

Second, Metronome takes advantage of this decoupling and uses *time-based* scheduling of garbage collector work, where the collector (when it is active) runs at regularly scheduled intervals of regular sizes. This makes the impact of the collector on the application highly predictable and reliable. By contrast, previous approaches typically performed *work-based* collection, where a unit of collector work was performed in response to application activity, for instance for every 4KB of memory allocated. Since allocation is typically unevenly distributed, work-based scheduling results in uneven interruptions of the application.

Third, Metronome is engineered in such a way that its individual work quanta are extremely small, both in the average and worst case, to the point where its resolution is sufficiently high for the vast majority of real-time applications.

Metronome's time based scheduler operates by providing a *minimum mutator utilization* or MMU [8]. The MMU specifies a time window and a percentage of the CPU time within that window which must be allocated to the application (a.k.a. the mutator). For instance, an MMU of 70% with a 10 ms time window means that for *any* 10 ms time period in the program's execution, the application will receive *at least* 70% of the CPU time (i.e., at least 7 ms).

A naive application of this approach would result in a maximum pause time of $1 - MMU$ times the window size (e.g., 3ms in the example). Also, the MMU contract is vulnerable to overshooting if the collector mis-estimates the time for a piece of work. This is solved by *overclocking* the collector scheduler. The collector runs at a nominal quantum size of 500 μs, and those quanta are spread out over the MMU window. Thus for the example above, we would nominally perform 6 separate quanta to perform the 3 ms of collector work. Those quanta are evenly spread, so that the application experiences less and more evenly distributed collector-induced jitter. Furthermore, the scheduler is able to do a much better job of guaranteeing the MMU target, since if a single quantum is larger than expected, it can simply either delay the next quantum slightly, or else perform a shorter quantum the next time.

In order to bound the work of scanning or copying an array (one aspect of bounding collector pauses), and also to bound external fragmentation caused by large objects, Metronome relies on *arraylets*, a technique for breaking large arrays into chunks that can be processed in bounded time. In the arraylet object model, all arrays consist of two parts: a *spine*, which contains the object header and pointers to *arraylets*, which are contiguous fixed-size chunks. Using 2^n sized chunks allows the indexing calculation to be done with shift and mask operations.

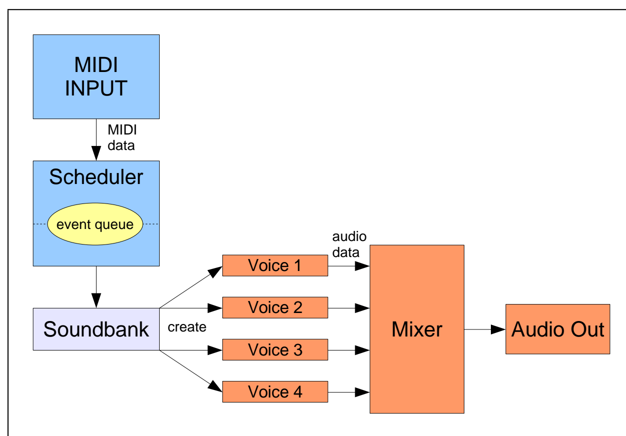

Figure 1. High-level Synthesizer Architecture

4. A JAVA-BASED SYNTHESIZER

The implemented system (code named *Harmonicon*) is a real-time software synthesizer written in pure Java. It implements the SoundFont 2 standard, which defines a synthesis engine based on wavetable samples. Real-time MIDI signals and/or Standard MIDI Files (SMF) are used to trigger notes for playback. The rendered audio stream is played on the computer's soundcard.

4.1. SoundFont 2 Synthesis

The SoundFont 2 [10] standard is based on a wavetable oscillator. For playback of a note, a recorded audio sample is processed and mixed together with other playing notes, then sent to the audio device. The processing step includes pitch shifting to adjust the pitch of the sample to the played note, low pass filtering, envelopes for volume, pitch, and filter and low frequency oscillators (LFO) for volume and pitch. Furthermore, many parameters can be changed in real-time to affect future notes and also currently playing notes.

In order to not compromise audio fidelity, internal processing is done with 64- bit floating point precision (Java data type `double`). Output rendering can be done at an arbitrary sampling rate and with up to 32-bits per sample. Pitch shifting is done with a sampling rate converter with linear interpolation. "Note stealing" (premature termination of notes due to limited polyphony) is avoided by allowing arbitrary polyphony. Currently, there is no mechanism to limit polyphony to match the available pro-

cessing power. Informal tests showed that the system supports more than 800 concurrent sustained notes on standard PC's.

The implementation allows very low latency and jitter to provide responsive and accurate music performance. Latency is mainly affected by the audio buffer size used in the rendering engine, but also by MIDI and audio hardware, and the quality of its drivers. The ALSA output plugin uses a double buffering scheme, where one buffer is played while the other buffer is rendered. The internal rendering quantum can be arbitrarily set; in the current configuration, it equals the buffer size if less or equal than 1 ms. Otherwise, multiple quantums are rendered per buffer, using a size close to 1 ms. To keep jitter down, we use forward synchronous timing [6] allowing the notes to be inserted with sample accurate timing into the rendered audio stream. However, when buffer sizes are very small we provide the option to insert notes as soon as possible; we compare the two approaches in Section 5.

4.2. Modularity and Flexibility

The implementation abstracts and encapsulates the logical units like MIDI Input, audio output, audio mixer, and the different units of the rendering engine so that "rewiring" and extending the system is easily possible. Parametrization of a wide range of aspects of the synthesizer allows fine-grained adjustments of quality, performance, and features. Most modules and settings can be changed at runtime during operation of the synthesizer. The modularity and flexibility was used for comparing measurement results with different settings or alternative implementations to evaluate performance vs. quality trade-offs.

The implementation extensively uses Java's features like object orientation, threading/locking, garbage collection, and the class libraries. This allows a code base that is safe, portable, and easy to maintain. We explicitly avoided programming constructs that introduce complexity or that reduce flexibility and modularization.

4.3. Advanced Features

Drift compensation is used to synchronize the MIDI time stamps provided by MIDI driver and the computer's real time clock to the audio card clock.

The engine supports arbitrary number of channels, it can be easily extended with support for 3-D and multitrack features.

The rendering engine can be set to use multiple threads, leveraging multiple CPU cores.

For very low buffer sizes, note scheduling can be done asynchronously in order to not disrupt the audio rendering loop with possibly computationally intense note initialization.

For low latency and low jitter, we implemented low-overhead native libraries for MIDI port input and audio output.

5. EXPERIMENTAL EVALUATION

We now evaluate the latency, jitter, and audio quality of our Java-based synthesizer running on top of our real-time Java virtual machine. The evaluation consists of two parts: first, we run the synthesizer at a range of buffer sizes (using both the Metronome-based real-time JVM and a standard non-real-time JVM) to determine how small a buffer size they can sustain with error-free performance.

We then perform absolute latency and jitter measurements (using the shortest buffer size sustaining error-free performance), and compare our system against a Kurzweil K2000R synthesizer. We also break down the latency and jitter contributions of the various components of the system.

5.1. Experimental Environment

All measurements of our synthesizer were done on an IBM Intellistation A Pro workstation with dual AMD Opteron 250 processors running at 2.4 GHz with a 1 MB level 2 data cache and a total of 4 GB of RAM. MIDI and audio interfacing was done via an M-Audio Audiophile 2496 sound card.

The operating system was IBM's real-time variant of Redhat Enterprise Linux (RHEL4 U2), a version 2.6.16 kernel in the `rtj12.11smp` configuration, which includes the HRT timers and the `PREEMPT_RT` patch. The ALSA sound drivers were version 1.0.14rc3, with a patch in the mpu401 module since they originally crashed the dual processor AMD machine. We used the ALSA library to access the `hw:` device directly.

The Java virtual machines were IBM WebSphere Real Time Version 1, service release 1, with the Metronome garbage collector (denoted "Metronome" in the figures) and the standard IBM Java 5 virtual machine, service release 4. The latter was run with three different garbage collection policies, selectable via the command line. In default policy (denoted "Stop-the-world" in the figures), the entire heap is collected when allocation would otherwise fail. In the "optimize average pause" policy (denoted "Incremental" in the figures), a concurrent thread is used to perform marking of the global heap partly in parallel with the application, minimizing garbage collection pauses. In the "generational concurrent" policy (denoted "Generational" in the figures), a "nursery" for short-lived objects is added to the incremental approach. The nursery can be collected more quickly than the global heap, with longer-lived objects promoted to a global heap when necessary. Class loading during synthesis was avoided by synthesizing empty notes, and by explicitly referencing a few classes using Java's Class.forName facility, at startup. All runs on the real-time JVM were performed using ahead-of-time compiled code.

5.2. Synthesis versus GC and Buffer Size

To evaluate the ability of a synthesizer written in Java to perform music synthesis we ran a series of experiments in

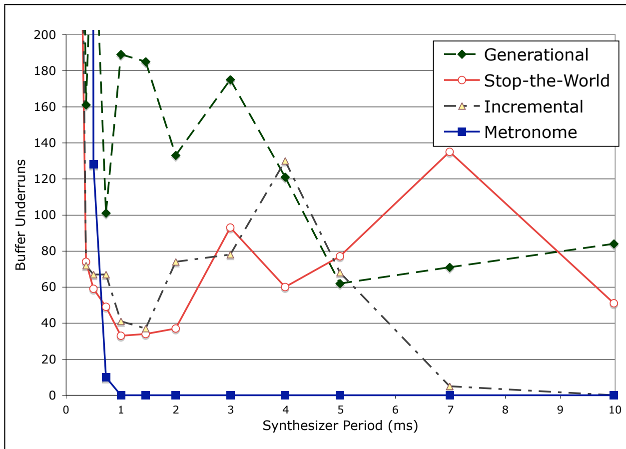

Figure 2. ALSA-reported Buffer Underruns

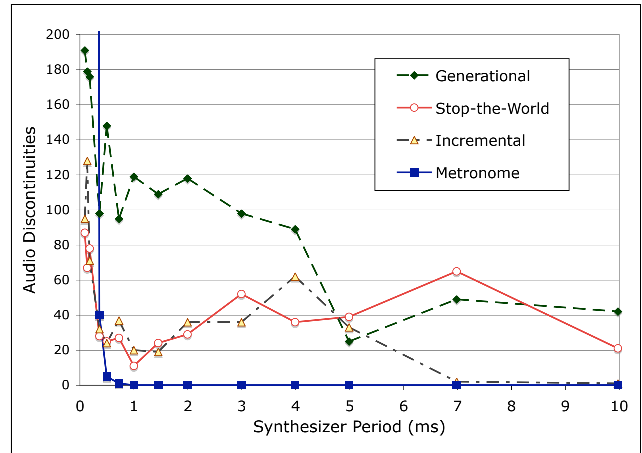

Figure 3. Waveform Discontinuities Detected in Recorded 44 KHz Audio

which the system rendered the identical sequenced MIDI file to the sound card.

The fundamental limiting factor in the system's latency is the size of the audio buffer which it renders each time through its loop, or the "period" of the system. We ran experiments using buffers from 90μs (4 samples) up to 10ms, under both the Metronome-based and the Standard JVMs.

In order to stress the system, we also created an additional thread which allocated data at 8 MB/s, comparable to what might be created by a compute-bound, highly object-oriented program running concurrently. We measured the system both with and without the memory load thread, but we will mostly report measurements with memory load since that stresses our technology much more heavily.

All measurements are for 34-second runs in which the beginning of Debussy's *Doctor Gradus ad Parnassum* is rendered by the synthesizer on the *Piano 1* instrument of a Creative Labs 8MB soundbank. The piece has a maximum instantaneous polyphony of 13 voices. The system is generating 2-channel audio with 32-bit resolution at 44.1 KHz.

We cross-validated our results by using two ways of measuring errors and glitches in the synthesized audio. First, we counted the number of buffer under-runs reported by the underlying ALSA audio driver, as shown in Figure 2 (memory load thread is active). The Standard JVM (regardless of garbage collection policy) behaves as builders of music software have come to expect from Java. For the stop-the-world and generational policies, there are tens of under-runs at buffer sizes from 500μs out to 10ms and beyond, making the JVM unsuitable for live audio processing. The incremental policy does best, but achieves acceptable performance only at the largest of the buffer sizes.

On the other hand, with Metronome there are no under-runs at all down to buffer sizes of 1ms (44 samples). At 0.73ms (32 samples), there are a small number of under-runs, and below that the number of under-runs grows very rapidly. At very small buffer sizes, Metronome generates more under-runs, due to the fact that the system is becoming CPU-bound and the real-time JVM makes some latency/throughput trade-offs.

Garbage collection behavior is quite clearly implicated in these results. During the run with a 1ms buffer, the Metronome collector performed 53 garbage collections, broken into 12,505 individual quanta that averaged 574μs with a standard deviation of 91μs and a maximum of 1.565ms. The small number of collector quanta over 1ms are absorbed by the 1ms of buffering.

In contrast, the standard JVM with the stop-the-world policy had 15 garbage collections, not quantized. The mean duration was 28.94ms with a standard deviation of 12.05ms. With the incremental policy, there were 5 "world-stopping" garbage collections with a mean duration of 9.11ms (standard deviation 2.51ms). There were also 11 partially concurrent garbage collections whose potentially interfering portions had a mean duration of 4.52ms (standard deviation 1.04ms). With the generational policy, there were 163 nursery collections (mean 8.22ms, std. dev. 2.87ms), 2 "world-stopping" global collections (10.13ms and 16.10ms), and 11 partially concurrent collections. The large number of nursery collections and their relatively long duration explains why the generational policy does worse than the incremental one (despite the fact that it is also partially concurrent). The memory load thread's allocation pattern included a fraction of longer-lived objects, which somewhat biased against a generational approach. Since we were interested in worst-case behavior, we believe this was nevertheless a fair test.

Under-runs reported by the ALSA driver do not always correspond to actual glitches in the audio (possibly, in those cases the under-run recovery was quick enough to prevent a completely empty hardware buffer). We also wished to verify the quality of the generated signal independently. Therefore, in each run we connected the line output of the synthesizer from the sound card to another sound card in another computer, recorded the waveform, and ran an analysis that examined the audio signal for artifacts that would result from dropouts. In particular, we

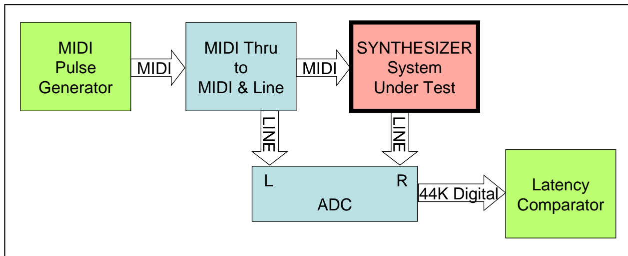

Figure 4. Experimental Design for Latency Measurements. Incoming MIDI signal is routed as an analog pulse and its transient is compared with the leading edge of the synthesized waveform of a percussion instrument.

	Min	Mean	Max	Std. Dev.
MIDI (C)	0.340	0.347	0.362	0.011
MIDI (Java Sound)	0.385	1.455	3.197	0.701
MIDI (Java/Direct)	0.385	0.406	0.430	0.011
Kurzweil K2000R	2.925	3.909	4.897	0.570
Metro 1ms	4.240	4.959	5.736	0.317
Metro 1ms/FSync	5.396	5.847	6.439	0.308
Metro 365μs/No GC	2.947	3.120	3.310	0.109

Table 1. Latency Summary

compute the power signal over small audio windows (22 samples or 0.5 ms) and look for abrupt drops (and subsequent recovery) in the power signal. The number of glitches as the buffer size is varied is shown in Figure 3.

The audio filter detects some glitches that are not indicated as under-runs, and conversely not all under-runs are detected as glitches. However, generally speaking, the results are in close agreement, and the Metronome-based synthesizer is also glitch-free down to 1ms.

These results match our informal listening tests: each audible artifact is also detected as a glitch and corresponds to a reported under-run. However, for some reported under-runs and glitches, we are not able to hear them. We suspect that they are too small to be audible, for example very few skipped samples.

5.3. Latency and Jitter

Having established that the Java-based synthesizer is able to produce clean audio running at a period of 1ms, we set out to determine its absolute end-to-end performance, determine the contribution of various portions of the system to latency and jitter, and finally, to compare it to an existing hardware synthesizer.

To perform the measurements accurately, we used a variant of the method developed by our colleague James Wright [23], as shown in Figure 4. A MIDI signal is sent from a computer acting as source to a Midi Solutions "Thru" box, modified to route its input signal out to an RCA jack as well as to the MIDI OUT port. The MIDI signal is then routed to the system under test.

We tested both MIDI-to-MIDI and MIDI-to-audio paths. For MIDI-to-audio, we synthesized a percussive instrument with a sharp attack. The system's LINE OUT signal is routed to the right channel of the sound input of another computer. The RCA output from the modified Thru box is routed to the left channel. Comparison of the leading edges of the two signals produces a quite accurate measurement of the total system latency.

For MIDI-to-MIDI latency, we used a program that simply echos all MIDI messages from the sound card's MIDI IN port to the MIDI OUT port, which was connected to a second modified MIDI Thru box, and also used left/right comparison to measure the total latency.

The results are summarized in Table 1. Echoing a 1-byte MIDI message from a C program took about $350 \pm 10\mu s$. Since MIDI [20] transmits one byte per $320\mu s$, this is quite fast. Furthermore, real synthesis will involve mostly 2- and 3-byte MIDI messages, adding at least $640\mu s$ to the total latency.

We first tried a similar Java program which used the standard Java Sound [17] APIs to read and write the MIDI messages, but as can be seen, this produced unacceptable results with latencies up to 3.2ms. This can probably be attributed to Java Sound's polling implementation, which uses Thread.sleep(1) in between polling for new MIDI messages. As a result, we wrote our own Java MIDI input layer, which calls the ALSA drivers directly via JNI. This improved things considerably: the Java program now only increases worst-case latency by $70\mu s$ over C. Java Sound is still used for MIDI output.

As a point of comparison for end-to-end performance of our system, we measured the widely used Kurzweil K2000R synthesizer, which we found to have an average latency of 3.9ms with 2ms peak jitter.

Running the synthesizer (*Harmonicon*) on the real-time JVM with a 1ms buffer and the memory load thread active, the system achieves 5ms latency with 1.5ms peak jitter — slightly slower but also slightly more stable than the K2000R. Overall, the two systems are roughly comparable in their achieved performance.

We also measured the effect of using forward-synchronous [6] scheduling of the arriving MIDI notes ("FSync"). While this produced a significant reduction in peak jitter, to 1ms, it also came at the expected expense of 0.9ms of additional latency.

Finally, to isolate the impact of garbage collection, we ran the system with the memory load thread off and a large enough heap that it never triggered a garbage collection ("No GC"), with the buffer size set to $365\mu s$ (16 samples), at which our previous tests indicated it could run without buffer under-runs. At this setting, the system improves significantly: 3.1ms end-to-end latency with only $360\mu s$ peak jitter.

6. RELATED WORK

6.1. Real-time Audio

Several real-time audio frameworks were developed in Java. They usually resort to a native language like C for the ac-

tual sample processing, and sometimes for event handling. This model reduces interruptions by the JVM, so real-time processing is possible to some degree. However, such an architecture introduces a number of drawbacks: in general, native libraries add security risks and greatly reduce portability. Java's ease of use cannot be taken advantage of; multi-threading, code maintenance and debugging are complicated. Last, but not least, the Java front end is still subject to the non real-time behavior of standard JVM's.

The JSyn [7] package provides a Java level API for creating interactive synthesis applications. All sample based processing is done in a native C implementation. A visual environment for creating interactive audio processing applications is jMax [12], building on the popular Max paradigm. Its Java GUI and the C realtime execution engine are separated using a hardware independent protocol. Both JSyn and jMax expose the drawbacks of a native audio processing layer.

A different approach is taken with Serpent [11], a high level scripting language and execution environment for interactive audio applications. It features a real-time garbage collector with less than 1ms pauses, and optimized thread context switches. Currently, the computational speed is slow, so its creator recommends interfacing it with modules written in C++ for sample based processing. While Serpent has properties that makes it attractive for creating interactive audio applications, its scope will be too limited for applications requiring the availability and versatility of Java.

SuperCollider [19] is a smalltalk-like language for computer music that also features a real-time garbage collector. As with Serpent, modules written in C++ are used for signal processing primitives, losing some of the advantages of a high level language.

6.2. Real-time Garbage Collection

There have been early attempts at providing real-time [3] or, at least, suitably responsive [22] garbage collectors to various environments. However, these systems were either impractical because of a flawed definition of real-time or the response was only often but not always acceptable.

Concurrent garbage collectors seek to achieve responsiveness by decoupling garbage collection activity altogether from the application. However, these collectors are either only mostly concurrent or suffer from memory fragmentation [13, 14]. Furthermore, when concurrency is the sole mechanism to achieve real-time response, the application's responsiveness is subject to the number of available processors and the operating system's scheduler.

Nettles and O'Toole [21] introduced replicating copying and provided an early collector in which actual responsiveness was fairly (but not guaranteed) predictable. Cheng and Blelloch [8] formalized the notion of MMU and presented a parallel, real-time collector that reliably met MMU goals in the 10ms range. Both of these collectors suffer from a large space overhead because of using a wholely replicating approach.

The Azul garbage collector [9] uses specialized hardware and does not decouple the mutator from the collector. Although most pauses are short, there are predictable periods of time when the application is unable to run at all for 20ms.

The Metronome [2] collector matched the earlier real-time replicating collectors in responsiveness while eliminating the space overhead. This prototype system was later developed into a production-quality system [1] with the addition of parallelism and other features we have already mentioned. It reliably meets MMU goals and achieves lower pause times ($<$ 1ms) than any earlier system.

7. CONCLUSIONS

We built a wavetable music synthesizer from scratch entirely in Java. We used this synthesizer as a hard real-time application to challenge the capabilities of the IBM WebSphere Real Time JVM, incorporating the Metronome garbage collector, running on RT Linux. We used Java's object-oriented features and a modular design, rather than employing programming styles typical of hard real-time applications of the past. Results show it is feasible to write an interactive music application entirely in Java. Our synthesizer has competitive end-to-end latency and jitter compared with a hardware synthesizer. Results also show that a real-time garbage collector such as Metronome is needed to achieve this favorable outcome.

8. REFERENCES

[1] AUERBACH, J., ET AL. Design and implementation of a comprehensive real-time Java virtual machine. Tech. rep., IBM Research, 2007. Submitted for publication.

[2] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proc. POPL* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, *38*, 1, 285–298.

[3] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[4] BOLLELLA, G., GOSLING, J., BROSGOL, B., DIBBLE, P., FURR, S., HARDIN, D., AND TURNBULL, M. *The Real-Time Specification for Java*. The Java Series. Addison-Wesley, 2000.

[5] BRANDT, E., AND DANNENBERG, R. B. Low-latency music software using off-the-shelf operating systems. In *Proc. of the International Computer Music Conference* (Ann Arbor, Michigan, 1998).

[6] BRANDT, E., AND DANNENBERG, R. B. Time in distributed real-time systems. In *Proc. of the International Computer Music Conference* (Beijing, China, 1999), pp. 523–526.

[7] BURK, P. JSyn: A real-time synthesis API for Java. In *Proc. of the International Computer Music Conference* (Ann Arbor, Michigan, 1998), pp. 252–255.

[8] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices*, *36*, 5 (May), 125–136.

[9] CLICK, C., TENE, G., AND WOLF, M. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments* (Chicago, IL, USA, 2005), pp. 46–56.

[10] CREATIVE LABS. *SoundFont 2.01 Technical Specification*, 1998.

[11] DANNENBERG, R. B. A language for interactive audio applications. In *Proc. of the International Computer Music Conference* (Gothenburg, Sweden, 2002).

[12] DECHELLE, F., BORGHESI, R., CECCO, M. D., MAGGI, E., ROVAN, B., AND SCHNELL, N. jMax: An environment for real-time musical applications. *Computer Music Journal 23*, 3, 50–58.

[13] DOLIGEZ, D., AND GONTHIER, G. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conf. Record of the Twenty-First ACM Symposium on Principles of Programming Languages* (Jan. 1994), pp. 70–83.

[14] DOMANI, T., KOLODNER, E. K., AND PETRANK, E. A generational on-the-fly garbage collector for Java. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 2000). *SIGPLAN Notices*, *35*, 6, 274–284.

[15] FULTON, M., AND STOODLEY, M. Compilation techniques for real-time Java programs. In *Proc. of the International Symposium on Code Generation and Optimization* (2007).

[16] IBM CORP. *WebSphere Real-Time User's Guide*, first ed., 2006.

[17] Java Sound API. http://java.sun.com/products/java-media/sound/.

[18] LUNNEY, H. Time as heard in speech and music. *Nature 249* (1974), 592.

[19] MCCARTNEY, J. Rethinking the computer music language: Supercollider. *Computer Music Journal 26*, 4 (2002), 61–68.

[20] MIDI MANUFACTURERS ASSOCIATION. *MIDI 1.0 Detailed Specification*, 4.2 ed. MIDI Manufacturers Association, La Habra, California, 1996.

[21] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices*, *28*, 6, 217–226.

[22] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, 1984), P. Henderson, Ed. *SIGPLAN Notices*, *19*, 5, 157–167.

[23] WRIGHT, J., AND BRANDT, E. System-level MIDI performance testing. In *Proc. of the International Computer Music Conference* (La Habana, Cuba, 2001).