An In-Depth Analysis of Real-Time MIDI Performance

Belinda Thom and Mark Nelson
Harvey Mudd College, Computer Science Department
{belinda_thom,mark_nelson}@hmc.edu

Abstract

Although MIDI is often used for computer-based interactive music applications, its real-time performance is difficult to generally quantify because of its dependence on the characteristics of the given application and the system on which it is running. We extend existing proposals for MIDI performance benchmarking so that they are useful in more realistic interactive scenarios, including those with high MIDI traffic and heavy CPU load. Our work has resulted in a cross-platform freely-available testing suite that requires minimal effort to use. We use this suite to survey the interactive performance of several commonly-used computer/MIDI setups, and extend the typical data analysis with an in-depth discussion of the benefits and downsides of various performance metrics.

1 Introduction

MIDI is a widely used standard for interconnecting electronic music devices and has both a communications protocol and a physical layer. It was originally designed to provide low-latency transmission of musical messages between devices, although arguments questioning its appropriateness in highly interactive real-time settings have been made (Wessel and Wright 2000; Moore 1988). Quantifying MIDI's latency is crucial because even small timing variations can be musically perceptible, especially when grace notes or other short ornaments are present. Researchers have proposed values as low as 1 to 1.5 milliseconds as an acceptable range of latency variation (Moore 1988; Wessel and Wright 2000), and around 10 milliseconds as an acceptable upper bound on absolute latency (Wessel and Wright 2000; Brandt and Dannenberg 1998).

Given MIDI's fixed 31.25 kHz baud rate, when connecting stand-alone synthesizers and sending messages of fixed size, associated communication delays are trivial to calculate, consistent, and relatively small. Our concern in this paper is with latencies that arise when MIDI communicates with software running on a general-purpose computer. Toward this end, we use *system* to refer to a general-purpose computer and all of its relevant interconnected parts: the MIDI interface and related drivers; the physical bus to which the interface is connected (USB, PCI, etc.); the operating system (including its scheduler, its MIDI API, and so on); and a specific run-time configuration (system priorities, power options, etc.). These system parts introduce additional latencies that are typically greater and less consistent than those associated with MIDI's physical layer. Nonetheless, MIDI's low cost and ready availability make it a frequent choice of researchers building interactive music systems (Biles 1998; Franklin 2001; Dannenberg et al. 2003).

Quantifying a system's latency is heavily dependent on the particular application. For example, as music researchers increasingly rely on more computationally expensive artificial intelligence techniques to proceduralize "musically reasonable" behavior, it becomes increasingly important to understand how processor load impacts latency. The amount of MIDI traffic is also likely to impact performance. For instance, an application's ability to accurately time-stamp incoming MIDI data could very well degrade when simultaneously sending out a steady stream of chordal accompaniment (our empirical data indicates that this is in fact a problem).

Our interest in quantifying system performance in such realistic settings was sparked by our desire to develop rhythm quantizers that could transform short segments of improvised notes into "appropriate" rhythmic notations in real time. One thing that sets our task apart is that we want to develop technologies that customize their mappings so as to "best notate" spontaneously generated rhythms in musician-specific ways. Recent advances in probabilistic modeling provide fertile ground for such user customization (Cemgil and Kappen 2001, 2003), but the iterative and approximate nature of these methods leads to their heavily loading the processor. Probabilistic models also provide disciplined ways for reasoning about uncertainty, and it was in thinking about this that we realized it was not at all clear what "error bars" we should use to model the accuracy of the time stamps that the computer assigns to incoming MIDI data during live performance. It was at this point that we took a step back and became interested in real-time MIDI performance testing.

Clearly, benchmarks for quantifying latency in realistic interactive music situations would be enormously valuable. Unfortunately, MIDI performance in realistic systems is typically poorly documented, and when it is empirically measured, the environment in which it is tested is often quite restricted. For example, Wright and Brandt (2001, 1999) provide a method for measuring a system's latency that is notably *independent*, by which we mean that quantification depends on an independent piece of hardware (as opposed to the system-under-test's clock). These tests, however, were performed using single active sense messages, no processor load, and with proprietary software generating the response.[1] A more complete (albeit dated) analysis of latency in off-the-shelf operating systems under various loads and configurations was done by Brandt and Dannenberg (Brandt and Dannenberg 1998), but their measurements rely on the system-under-test's notion of time. To address these deficiencies, we have developed a freely-available cross-platform software package that, when used in conjunction with the inexpensive and easy-to-build *MIDI-Wave transcoder* circuit proposed by Wright and Brandt, can be used to independently test the performance of a particular system *in-place*. The software and accompanying documentation are available online.[2]

The work presented here is important in part because a myriad of factors can influence real-time system performance. Thus it becomes desirable, and in some cases essential, for researchers—particularly those developing interactive MIDI applications—to be able to quantify performance *for their particular application and system.* To increase the odds that our test package will be applicable to the general public, we significantly extended upon the methodologies used by Brandt, Dannenberg, and Wright. For example, in addition to active sense data, we developed more realistic *burst* and *load* tests. Burst tests are interesting because multiple MIDI note messages are periodically transmitted in groups, producing the type of situation that arises when playing real-time background accompaniment. Load tests do the same thing under extensive CPU load, a likely scenario when generating interactive accompaniment on-the-fly. In important selling point of our tests is that they are based on *PortMidi*, a free, lightweight, cross-platform MIDI library,[3] which means that users who run our tests can easily migrate from testing to writing their own PortMidi-based applications.

We have used our methodology to survey the performance of several popular MIDI interfaces on three major consumer operating systems (Linux, Mac OS X, and Windows). While these tests are certainly not exhaustive, they illustrate the many issues involved in quantifying and analyzing real-time MIDI performance and serve as useful points of reference. An overview of these performance results has already been published (Nelson and Thom 2004). The purpose of this paper is to significantly extend upon this prior work, exploring in detail the benefits and pitfalls of our testing methodology, analysis techniques, and statistical measures. Our hope is that this work will enable more members of the community to rigorously quantify and tune their systems' performance. As ad hoc performance tweaks are replaced by systematic benchmarking methods, everyone benefits, for the methods themselves can evolve and adapt, becoming even more useful.

[1]Details from Jim Wright, personal communication.

[2]`http://www.cs.hmc.edu/~bthom/downloads/midi/`

[3]`http://www.cs.cmu.edu/~music/portmusic/`

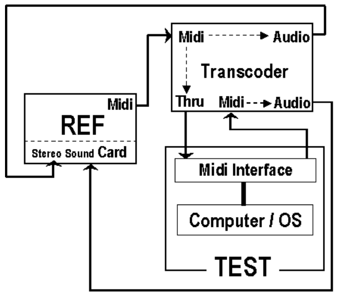

Figure 1: Overview of our MIDI performance test setup.

2 Methodology

Our empirical testing extends the MIDI-Wave transcoder methodology proposed by Wright and Brandt (2001, 1999), adding many more types of tests and real-time data analysis. A schematic overview of the test setup is shown in Figure 1.

2.1 The Midi-Wave Method

Each test involves two systems: The reference system (*REF*) generates a stream of MIDI messages that are presented to the system whose performance is being tested (*TEST*). TEST, running a simple PortMidi application, forwards the REF stream MIDI messages back out, producing the TEST stream. The Midi-Wave device sits between the two systems, transcoding a copy of each stream into audio. The TEST and REF audio signals are recorded in stereo via the REF system's sound-card line-in. Latency is measured by our analysis software, which runs on the REF system and compares the delay between the two audio streams in real time. The transcoder allows us to use the sound-card as a cheap, readily-available two-channel voltage sampler, as transcoded audio is nothing but a raw MIDI signal, converted into an appropriate voltage range. Sample audio is shown in Figures 2 and 3.

Recording a 31.25 kHz MIDI signal at the standard 44.1 kHz audio sampling rate suffices because we are merely interested in locating the positions of MIDI messages in a stream. In particular, the Nyquist Theorem, which would recommend sampling at 62.5 kHz, does not apply because we are not interested in reproducing the signal digitally. Rather, our sampling rate must simply be fast enough to not miss bit flips. With a 31.25 kHz MIDI data rate, a MIDI bit spans approximately 31 μs. 44.1 kHz, which provides a 23 μs sampling period, ensures that no bit flip will be dropped. As detailed elsewhere (Wright and Brandt 1999), latency measurements accurate to within 0.1 to 0.2 ms are easily obtained.

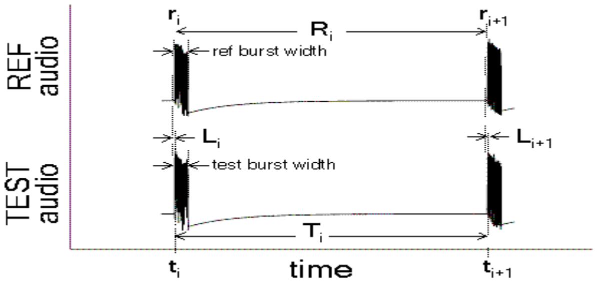

Figure 2: Sample transcoder audio (from a G4 OSX 2x2 burst test). The variables in this figure are defined in Section 3.

2.2 Modifications and Proposed Benchmarks

Since the proprietary *Cubase* sequencing software was crafted at a very low level to provide improved performance, Wright and Brandt's tests ran this software on their TEST systems.[4] In contrast, we are more interested in testing under conditions similar to those that an application-level software developer might encounter. This goal led us to develop a light-weight PortMidi application for our TEST systems that, in a high-priority background thread, simply checks for MIDI input once per millisecond, forwarding messages on via MIDI out. In the foreground, a main thread runs that either periodically loads the CPU or sleeps, depending on what type of test is being run.

We have also significantly extended the tests that can be run. Wright and Brandt's original tests only sent one-byte active sensing messages at fixed intervals. Active sense messages do bring a consistency to testing because they are not typically treated in special ways by drivers or operating systems. Alone, however, the MIDI traffic patterns they generate are unrealistic. Thus, in addition to active sensing, we added a number of other ways for users to vary MIDI traffic patterns. For example, note-on and note-off messages can be generated; messages may be sent either individually or in bursts of varying size; and messages or bursts of messages may be output at user-specified frequencies. There is also an option to run a test under simulated load (arbitrary arithmetic on a 1-megabyte matrix), producing approximately 100% CPU utilization and memory usage sufficient for clearing the CPU cache.

From the many possible combinations of these options, we chose three tests as benchmarks:

- ***sense***: one active sensing message every 35 ms.
- ***burst***: bursts of ten note-on/off messages every 100 ms.
- ***load burst***: same as ***burst***, but loaded TEST system.

We ran each test for an hour, a duration arrived at through some empirical testing. Short (e.g. 15-second) tests can characterize average performance reasonably well, so are useful as quick indications, but performance problems on some systems show up only occasionally. For example, in some of our tests worst-case performance over an hour was 5–7 ms worse than worst-case performance over 15 seconds. Although even longer tests may indicate still more rare instances of performance degradation, we did not see such degradation in the few 10-hour experiments we ran.

2.3 Real-Time Analysis

A key feature of our test suite is its real-time analysis. Without real-time analysis, an hour-long test would require recording and analyzing 600 MB of audio data. For those users whose interest in highly reliable determination of worst-case latency makes tests on the order of 10 hours desirable, the prospect of recording and analyzing 6 GB of data is even less appealing!

Our real-time analysis uses a relatively simple thresholding algorithm to locate message "groups"—either single active sense messages or bursts of multiple messages—in each stream. Groups in the REF stream are matched up with their corresponding groups in the TEST stream and corresponding groups are compared to calculate latency and width.

Our thresholding method requires that message groups be well-separated. That is, the frequency with which groups are sent must be low enough so that the end of one does not get too close to the beginning of another; otherwise, it is difficult to determine where one ends and another begins.[5] Wright and Brandt must have used a more complex signal analysis scheme (perhaps autocorrelating over the entire stream), for they analyzed audio data collected for active sensing messages sent every 4 ms (yet their reported maximum delays ranged from 4.2 to 17.9 ms). We played with various periods for active sensing, settling on 35 ms because it robustly separated message groups on all of our test systems. Although our simple threshold scheme is restrictive—e.g. 35 ms event spacing is about 2% of MIDI 1.0 DIN capacity, whereas 4 ms is 25%—the benefit is real-time analysis, which allows us to run tests of arbitrary length. Another benefit of our algorithm is that analysis errors are very unlikely to occur because we require that each REF and TEST group match and that no detected latency be larger than the generating period. As a result, errors in detecting thresholds will almost certainly produce failed tests as opposed to faulty data.

Real-time audio analysis is implemented using the cross-platform PortAudio toolkit.[6] In fact, both PortAudio and

[4]This and many other details were confirmed with Jim Wright in personal communication.

[5]To help debug in cases where bursts "stack up," our software dumps an audio file containing audio recorded over the past few problematic periods. Since analysis is done in real-time, this kind of dump is required if problematic data is to be accessed.

[6]PortAudio (Bencina and Burk 2001) performs a similar function for audio that PortMidi does for MIDI; it is freely available at

PortMidi libraries contribute to the REF system's main program.

2.4 System Configurations Tested

We tested a selection of systems composed of the commonly-used components listed below. Italicized abbreviations will be used when reporting results for particular systems.

Interfaces:

- **Midiman MidiSport** *(2x2)*, USB
- **MOTU Fastlane** *(Motu)*, USB
- **EgoSys Miditerminal 4140** *(4140)*, parallel port
- **Creative Labs SoundBlaster Live! 5.1** (*SB* or *SBLive*), PCI sound-card with integrated MPU-401 compatible interface

Operating systems (and their MIDI APIs):

- **Linux with 2.4-series kernel** *(Linux 2.4)* using the Debian GNU/Linux distribution with ALSA 0.9.4, kernel 2.4.20, and some low-latency patches.[7]
- **Linux with 2.6-series kernel** *(Linux 2.6)*, as above but with ALSA 0.9.7, kernel 2.6.0, and no special patches.
- **Mac OS X** *(OSX)* 10.3.2 (Panther) with CoreMIDI.
- **Windows 2000** *(Win2k)* SP4 with WinMME.
- **Windows XP** *(WinXP)* SP1 with WinMME.

Computers:

- **HP Pavilion 751n desktop** *(HP)* with 1.8 GHz Intel Pentium 4 processor and 256 MB RAM.
- **Apple Mac G4 desktop** *(G4)* with dual 500 MHz G4 processors and 320 MB RAM.
- **IBM Thinkpad T23 laptop** *(T23)* with 1.2 GHz Intel Pentium II processor and 512 MB RAM.

A few notes on configuration: We made an effort to ensure that the systems were configured reasonably, but given the range of possible configurations, there is likely still room for improvement (in fact, our tests can be used to help guide the search through this configuration space!). Under Windows, MIDI was handled by a multimedia thread, with system priorities set as recommended on the PortAudio website.[8] Under Linux, the MIDI-handling thread ran with `nice` value -19.[9] With OS X, the MIDI-handling thread ran as a fixed-priority, non-time-sharing thread with precedence 30. Under all operating systems, when load tests were run, the main loading thread had default priorities. We also took other reasonable steps to enhance performance: turning off virus scanners, disabling network access, disabling power saving features (hard drive spin down, screen-savers), and so on.

http://www.portaudio.com.

[7]Robert M. Love's variable-Hz (Hz=1000) and pre-emptible kernel patches and Andrew Morton's low-latency patch.

[8]http://www.portaudio.com/docs/latency.html

[9]The software must be run as root for this heightened priority.

Previous tests (Wright and Brandt 2001) suggested that USB interfaces, which are newer but quickly becoming the de facto standard, perform more poorly than "legacy" interfaces (parallel or serial port, PCI), so we tested both types. We did not test FireWire because of their steep prices (over US$500 as of this writing).

Not all interfaces could be tested on all operating systems. OS X's CoreMIDI only supports USB. With Linux, no 4140 drivers could be found. We made numerous attempts to get the Motu to work on Linux but were never successful. Additionally, the early revisions of Linux 2.6 available at the time of testing display USB problems on some hardware. For this reason, the 2x2 was only tested on Linux 2.4. For OS X and Windows tests, we used the newest drivers available as of November 2003 on their manufacturers' websites. As no manufacturers provide Linux drivers, reverse-engineered open-source drivers were used.[10] Finally, since the Motu would not forward active sense messages on OS X, note-on/off messages were used for both USB interfaces on Mac machines.

3 Terminology and Statistics

Typically, two terms are used when characterizing system-induced MIDI delay. *Latency* is usually defined as the delay introduced by a system when transmitting a MIDI message, whereas *jitter* is how much this delay varies over time.

It is easy to use these terms ambiguously because both intimately depend on the way in which delays are distributed. For instance, although each count stored in a latency histogram refers to a *specific* event's delay, aggregate statistics that summarize a collection of events are often used (Wright and Brandt 2001; Brandt and Dannenberg 1998) to quantify results: e.g. *average latency*; *worst-case latency*; *peak jitter*, which is the difference between minimum and maximum observed latency values; etc. In another "latency" definition (Brandt and Dannenberg 1998), measurements were calculated by first taking the difference in time between adjacent timer call-backs and then subtracting off the constant period in which the timer was scheduled to run.[11] We will soon provide a framework that shows the relationship between this use and the prior latency definition. Fortunately, regardless of the individual nuances in terms, everyone seems to agree that the *distribution* that describes system delays is the primary quantity of interest.

In Figure 2, transcoder data for two bursts of MIDI, i and $i+1$, is displayed. REF burst start times, r_i and r_{i+1}, correspond to TEST start times t_i and t_{i+1}. Two latency measurements result: $L_i = t_i - r_i$ and $L_{i+1} = t_{i+1} - r_{i+1}$. In

[10]The `emu10k1` ALSA driver for the SBLive, and the `usb-midi` driver for the 2x2.

[11]WinMME timer resolution cannot exceed 1 ms. Microsoft used a similar scheme for measuring latency in sample code shipped with *Visual C*.

our tests, each burst i contributes one latency measurement to a *Transcoder Latency* histogram. An example histogram is displayed in Figure 4). This histogram's main power is its quantification—in absolute terms—of system responsiveness. In terms of aggregates, worst- and average-case L_i estimates are probably the most useful. In the spirit of Brandt and Dannenberg (1998), transcoder data can also be used to collect period-based measurements; for example, REF period $R_i = r_{i+1} - r_i$ and TEST period $T_i = t_{i+1} - t_i$. Period-based quantities become more useful when the goal is to recreate a stream of periodic inter-onset intervals (IOIs) as closely as possible. As we will show below, a better measure of periodic fidelity is a *Transcoder δ Latency* histogram, which is constructed from $\delta L_i = L_{i+1} - L_i$, the difference between adjacent latencies. Again, see Figure 4 for an example. Note that a δL_i histogram cannot be calculated after the fact from an L_i histogram alone as the compressing act of binning data throws away crucial temporal information.

By definition:

$$L_i + T_i = R_i + L_{i+1}. \qquad (1)$$

Since $r_i < t_i$ and $r_{i+1} < t_{i+1}$, this relationship directly follows (see Figure 2). If L_i was normally distributed with standard deviation σ, Equation 1 would predict δL is also normally distributed with standard deviation $\sqrt{2}\cdot\sigma$. This increase makes sense: Adding two identically distributed random variables increases the overall sum's uncertainty.[12] In our experiments, including the results shown in Figure 4, δL's empirical distribution is much less spread out than L's. Although a normal distribution is inadequate for modeling system delays—for example, latencies are strictly non-negative, distributions tend to be bimodal, and so on—the most likely reason for this unexpected decrease is that system latencies exhibit temporal dependence.

An important insight results by combining Equation 1 with δL_i's definition:

$$T_i = R_i - \delta L_i. \qquad (2)$$

In Equation 2, we see that when IOI fidelity is most important, the performance measure of interest is the δ Latency distribution. Another implication of this fact is that peak jitter will provide an overly pessimistic view when IOI fidelity is the main concern because it does not require that only adjacent latencies be considered. In contrast, when quick responsiveness to onsets is the key, maximum latency, in conjunction with peak jitter, are quite relevant. Equation 2 also links the "latency" measure that Dannenberg and Brandt used with our definition. In their case, the timer's ideal period was fixed, which amounts to setting each R_i to this constant value. Under the reasonable assumption that call-backs execute almost instantaneously (within a few μs), what is measured in this scheme is the δL_i distribution.

[12]The square root is a result of the fact that additive normal errors propagate via adding in quadrature.

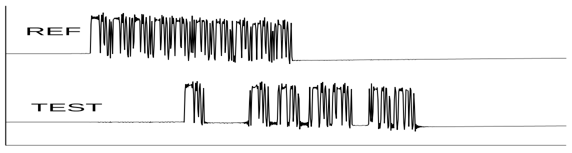

Figure 3: A REF burst is "stretched" as the TEST system is falls behind.

To quantify bursty real-time behavior, we record the *widths* of the audio transcoded for a burst of messages. Width, as shown in Figure 2, is the time between the beginning and end of a burst message. The distribution over width quantifies a TEST system's ability to receive and process bursts of MIDI messages in a timely manner. Figure 3 provides an example of how a bursty message can be "stretched" because the test system is unable to keep up with sending out the signal as soon as it comes in.

4 Results

A single test run produces a set of histograms like those shown in Figure 4. We focus here on this 4140-based system because its poor performance makes for interesting discussion.[13] Together, the Transcoder Latency and Test Width histograms provide a reasonable characterization of the system's absolute responsiveness and the Transcoder δ Latency histogram reasonably characterizes IOI fidelity. The much smaller values in the δ Latency histogram suggest a high degree of adjacent temporal dependence—while many L_is are around 20 ms, only two δL_is lie above the 4 ms range.

The TEST Periodic Timer histogram displays data collected by the TEST system entirely in software. This histogram captures the variability that the TEST system observed in servicing its 1 ms periodic MIDI-thru timer call-back. As opposed to an independent measure, this histogram's data is referenced with respect to the TEST system's internal clock. It makes sense that this histogram should correlate somewhat with the transcoder's, as obvious sources of MIDI latency include the operating system's ability to schedule things on time. Having said this, one might conclude that a purely software-based approach to performance testing would suffice; indeed, Brandt and Dannenberg used this method. However, as the plots in Figure 4 illustrate, the software histogram gives a much less accurate view of latency behavior than does the transcoder histogram, so we always recommend spending the extra effort needed to build such a device.

It is worth drawing attention to the difference in variability between the TEST and REF width histograms. The REF system (Linux 2.4, HP, SBLive) was *only* producing periodic bursts of output, and it was able to realize this behavior very

[13]Histograms and log files for all of the systems we tested are available on our website.

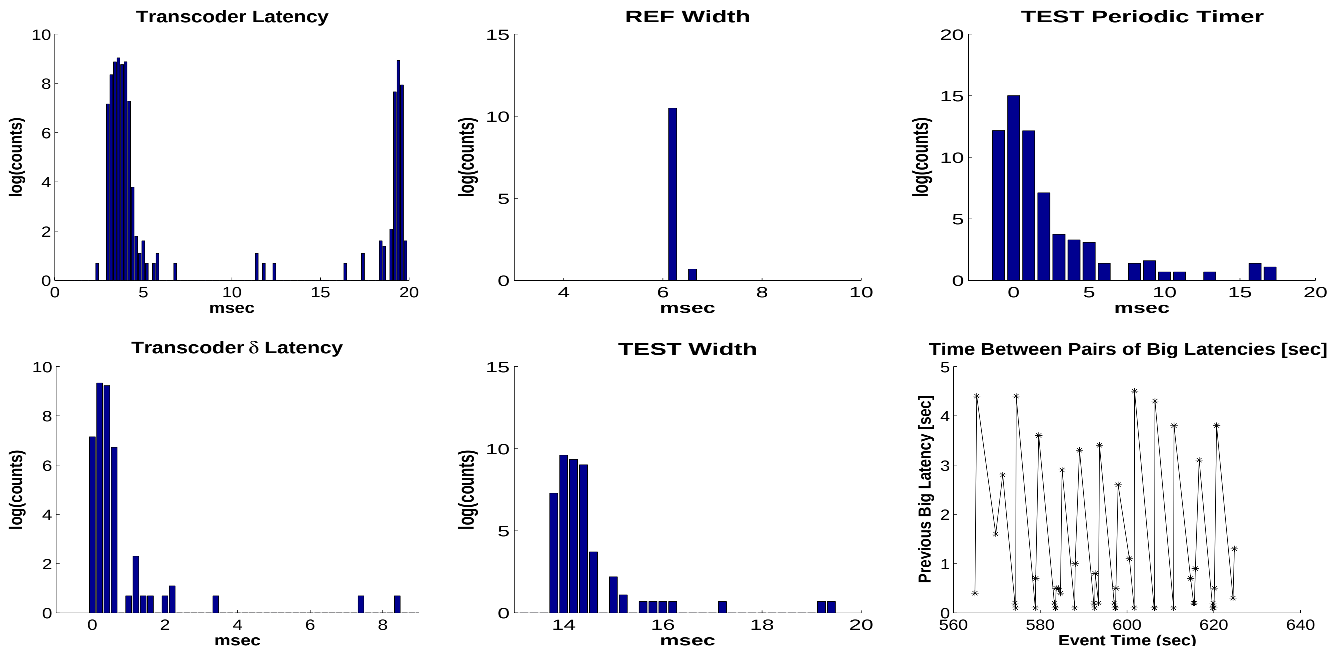

Figure 4: A sample selection of histograms (load burst on T23, Win2k, 4140), except for the lower-right-hand figure, which displays data taken from a load burst test on G4, OSX, Motu (see text for details).

consistently. The TEST system, which had to not only process asynchronous MIDI input but also send it back out, had a much more difficult time. We saw this kind of behavior on virtually every system we tested—including one in which the TEST and REF systems were identical. These results suggests that MIDI input is inherently more difficult to process in real-time. We thus recommend that bi-directional communication be a primary focus in performance benchmarking.

Various summary statistics that quantify the results of our performance survey are shown in Table 1. For brevity, only best-case (sense) and worst-case (load burst) tests are reported. For unloaded burst test results, and additional statistics and discussion, see our NIME paper (Nelson and Thom 2004). Realize that, while the summary statistics in this table are useful, they do obscure valuable information about the underlying distributions. For example, the Transcoder Latency histogram in Figure 4 is clearly bimodal; simple aggregate measures will never adequately characterize this fact. At the same time, histograms are not the end-all and be-all; they treat each count that is recorded as independent of all the others and so any kind of time dependence is thrown away. While it is true that the Transcoder δ Latency histogram contains temporal information regarding adjacent events, higher-order temporal effects are again lost.

Temporal latency dependence is substantiated in Table 1: δ latency aggregates are all less than their absolute latency counterparts. This observation motivates in-depth investigations of performance-related temporal issues. Our cursory investigation of this topic, however, has not borne much fruit, even though simple modifications to our tools support fairly open-ended exploration in this area. For example, by saving 5- to 10-minute streams of time-stamped latencies, we were able to generate the lower-right-hand plot in Figure 4.

This plot displays how problematic latencies—which we defined to be latencies greater than or equal to 7 ms—distribute over time. Test time in seconds proceeds along the x-axis. Each data point (asterisk) corresponds to a problematic latency. The y-axis simply reformats the information provided on the x-axis, making it easier to interpret: A data point's vertical location indicates how far away it was (in time) from the previous problematic event. Recall that bursts are sent out every 100 ms. Thus, data points with 100 ms y-axis values indicate two (or more) adjacent bursts serviced behind schedule. Clusters of such points indicate contiguous spans in which the system was having trouble keeping up. Another large population of problematic latencies are separated by 3 to 5 second spans. Unfortunately, the temporal distribution of problematic latencies in this figure does not display any “trivially systematic” structure. In short, we believe it would be very difficult to predict in advance exactly when such events will occur.

The good news for interactive MIDI applications is that the best-performing systems in our tests exhibit performance very close to the absolute targets of 10-ms latency and 1- to 1.5-ms jitter that we discussed in the introduction. The best overall performer in our particular setup—the SBLive on the HP desktop running WinXP—has in its worst-case results (the load burst test) a maximum latency of 2.8 ms, peak

System	Sense							Load Burst								
	msec							msec								
	μ_L	σ_L	p_L	m_L	$\mu_{\delta L}$	$\sigma_{\delta L}$	$m_{\delta L}$	μ_L	σ_L	p_L	m_L	$\mu_{\delta L}$	$\sigma_{\delta L}$	$m_{\delta L}$	p_w	m_w
HP Linux2.6 SBLive	0.8	0.3	2.1	2.3	0.0	0.1	1.4	1.2	0.3	7.0	7.6	0.0	0.1	6.6	2.4	8.6
HP Linux SBLive	0.8	0.4	25.4	25.6	0.0	0.1	24.7	1.2	0.4	26.0	26.6	0.0	0.1	25.9	17.7	23.9
HP Linux 2x2	2.2	0.5	24.7	25.7	0.0	0.2	24.0	3.7	0.5	34.4	36.4	0.0	0.2	32.8	21.6	29.0
G4 OSX 2x2	3.5	0.4	2.2	4.6	0.5	0.1	1.7	3.6	0.4	3.2	5.8	0.4	0.3	2.2	8.7	18.1
G4 OSX Motu	5.4	0.6	3.4	7.0	0.4	0.5	3.0	5.7	0.7	5.6	9.2	0.3	0.5	3.0	7.2	10.6
HP WinXP SBLive	0.9	0.3	2.0	2.4	0.1	0.2	1.3	1.3	0.3	2.0	2.8	0.6	0.2	1.7	1.2	10.6
HP WinXP 2x2	3.5	0.5	3.2	5.4	0.3	0.4	2.2	5.8	0.6	5.4	7.8	0.9	0.5	3.6	3.9	12.5
HP WinXP Motu	7.5	1.5	8.0	12.2	1.8	1.4	3.2	7.9	1.5	8.0	12.6	1.0	1.2	4.0	6.8	13.2
T23 Win2k 2x2	4.3	0.6	3.9	6.3	0.1	0.4	2.1	6.8	0.5	7.8	10.6	0.1	0.4	4.0	4.2	13.6
T23 Win2k Motu	7.7	1.3	5.1	10.3	1.0	0.5	2.2	7.7	1.2	5.0	10.6	0.1	0.3	4.9	8.4	14.8
T23 Win2k 4140	2.1	0.8	3.6	4.4	0.5	0.8	3.3	3.7	0.3	18.3	20.7	0.3	0.2	16.6	5.7	19.5

Table 1: Summary statistics for various tests. The empirical transcoder latency distribution is characterized by: mean (μ_L), standard deviation (σ_L), peak jitter (p_L), and maximum (m_L). The transcoder δ latency distributions are characterized by the same statistics, except that peak jitter is omitted (the minimum δL is zero in all cases, so peak jitter and the maximum are identical). For load burst tests, width is characterized by peak jitter (p_w) and maximum width (m_w).

jitter of 2.0 ms, and peak jitter in the burst widths of 1.2 ms, all very respectable figures.

The bad news is that none of the other configurations we tested exhibited performance at quite this level, at least when running the load burst tests. A common problem, exhibited by the otherwise admirably-performing 2x2 on the G4 running OSX, is fairly large width jitter in the load burst tests. Since all messages take some time to send, the peak jitter in width provides the most useful measure: 8 to 10 ms differences can be expected when delivering bursts of messages on a G4 under load. Note that the G4's absolute latency values are on par in both the sense and load burst tests, suggesting that the most problematic aspect of load is that it significantly delays notes occurring later on in the burst.[14] Perceptually, this behavior might lead to chords sounding slightly arpeggiated.

One pleasant result is that the performance of Linux 2.6 is vastly improved over that of Linux 2.4, especially in terms of maximum latency and peak jitter. Linux's performance for real-time tasks had previously been rather poor; the substantial efforts made by kernel and ALSA developers in addressing that criticism have obviously been successful. For our purposes, the new version of Linux is an ideal option, since it nicely complements the open-source model of PortMidi/PortAudio, and we can tolerate a 7-ms jitter. Similarly, those who can accept jitter in the 5- to 7-ms range can consider using the USB interfaces on OSX. This will be particularly useful if the G4 laptops perform similarly to the desktops (we're optimistic, given the similarity of the hardware).

The worst victim of system load is the 4140, which, while it outperforms the USB interfaces on a lightly-loaded system,[15] degrades very badly when tested under load, possibly a result of the way the low-level parallel port's hardware interrupts interact with the operating system. In the sense tests, on the other hand—where messages are kept relatively sparse without large bursts and there is minimal system load—about half the interfaces perform reasonably well, with peak jitter under 4 ms. The impact of this result is that we have yet to find a good solution for PC laptops, which do not support PCI sound-cards like the SBLive. We had originally purchased the 4140 in the hopes that a low-level parallel port interface would perform better than the USB alternative, but its poor performance under load makes it impractical. We emphasize this particular example because it powerfully illustrates the need to replace ad hoc guesses about performance with a rigorous set of tests.

It is worth emphasizing that the results reported here apply to *specific* systems. For example, because OSX only supports USB one could argue that it is unfair to directly compare results obtained for such a system with those obtained for an "equivalent" PCI WinXP alternative. At the same time, when building interactive music applications, the primary concern is often deliverable real-time performance. As long as the interfaces themselves tend to impact performance, what interfaces a given platform supports will remain an important consideration.

5 Future Work

Further modifications to our testing tools are worth exploring in order to simplify their use and increase the range of situations they can test. In particular, the constraint mentioned in Section 2.3—that message groups be well-separated—

[14]Recall that for our Mac sense tests, note-on/off messages were sent because the Motu would not forward sense messages on.

[15]Including in the unloaded burst tests not reported here.

would be nice to do away with. It has been suggested to us[16] that integrating a UART into the transcoder might allow us to convert each MIDI byte into a well-separated single spike. An extension like this would allow us to test periodic MIDI traffic at higher frequencies.

It is also worth exploring the "scheduled output" MIDI APIs found on some operating systems (e.g. the WinMME stream interface, which PortMidi supports). This technology allows a MIDI message to be scheduled for output at some point in the future, instead of being sent out immediately. By scheduling messages to be output in, say, 1 to 5 ms, high-priority scheduling might be passed of into the operating system kernel, where it may is more likely to be serviced consistently. This kind of behavior would allow applications to trade off an increase in latency for a decrease in jitter.

Finally, we would like to investigate how repeatable our tests are, for example, running the same test on a given system multiple times. One thing that such an investigation could explore, for example, is to what extent repeatable results require a freshly booted machine.

6 Conclusion

Although it turns out that MIDI can indeed perform close to the threshold of perceptible timing error, it is clear that performance can differ significantly, both due to the configuration of the system and due to the nature of the MIDI traffic. Furthermore, it is not at all obvious how to best quantify performance generally, given the different constraints present in different contexts. Previous performance testing did not bring all of these facts to light. We hope that our discussion and analysis will, in addition to illustrating some common sources of latency and jitter, encourage researchers using MIDI for interactive computer applications to use independent, in-place tools to test and tune the performance of their systems.

One of our hopes in developing this more realistic MIDI test suite is that it will foster active community participation. Certainly we are not the only ones who share this interest—existing resources such as Jim Wright's OpenMuse[17] have similar goals. Imagine, for example, the benefits of a resource where individual researchers could report and discuss empirical performance measures for their particular applications on specific systems. Such interaction would likely lead to a robust and generally accepted set of useful benchmarks for interactive music applications, as well as an extensive survey of system performance. This would be tremendously useful to those designing their own interactive music systems, as currently it is not at all clear which interfaces, operating systems, and configurations one ought to choose for various applications. In addition, it would provide a rigorous basis from which to evaluate the relative merits of various protocols that have been proposed as replacements for traditional MIDI, such as Ethernet MIDI and Open Sound Control.

[16]Roger Dannenberg, personal communication.

[17]`http://www.openmuse.org`

Acknowledgments J. Wright, E. Brandt, and R. Dannenberg provided invaluable guidance regarding MIDI performance testing. C. Baumgaertner and D. Harris advised us on matters electronic. M. O'Neill, C. Connelly, and G. Kuenning contributed Mac and Linux systems expertise. The Harvey Mudd Computer Science Department and Faculty Research program providing funding and a supportive environment.

References

Bencina, R. and P. Burk (2001). PortAudio – an open source cross platform audio API. In *Proceedings of the 2001 International Computer Music Conference (ICMC-01)*.

Biles, J. (1998). Interactive GenJam: Integrating real-time performance with a genetic algorithm. In *Proceedings of the 1998 International Computer Music Conference (ICMC-98)*.

Brandt, E. and R. Dannenberg (1998). Low-latency music software using off-the-shelf operating systems. In *Proceedings of the 1998 International Computer Music Conference (ICMC-98)*, pp. 137–141.

Cemgil, A. T. and H. J. Kappen (2001). Bayesian real-time adaptation for interactive performance systems. In *Proceedings of the 2001 International Computer Music Conference (ICMC-01)*, pp. 147–150.

Cemgil, A. T. and H. J. Kappen (2003). Monte Carlo methods for tempo tracking and rhythm quantization. *Journal of Artificial Intelligence Research 18*(1), 45–81.

Dannenberg, R., B. Bernstein, G. Zeglin, and T. Neuendorffer (2003). Sound synthesis from video, wearable lights, and 'The Watercourse Way'. In *Proceedings of the Ninth Biennial Symposium on Arts and Technology*, pp. 38–44.

Franklin, J. (2001). Multi-phase learning for jazz improvisation and interaction. In *Proceedings of the Eighth Biennial Symposium on Arts and Technology*.

Moore, F. R. (1988). The dysfunctions of MIDI. *Computer Music Journal 12*(1), 19–28.

Nelson, M. and B. Thom (2004). A survey of real-time MIDI performance. In *Proceedings of the 2004 Conference on New Interfaces for Musical Expression (NIME-04)*. In press.

Wessel, D. and M. Wright (2000). Problems and prospects for intimate musical control of computers. In *Proceedings of the ACM SIGCHI CHI '01 Workshop on New Interfaces for Musical Expression (NIME-01)*.

Wright, J. and E. Brandt (1999). Method and apparatus for measuring timing characteristics of message-oriented transports. United States Patent Application. Granted 2003, Patent 6,546,516.

Wright, J. and E. Brandt (2001). System-level MIDI performance testing. In *Proceedings of the 2001 International Computer Music Conference (ICMC-01)*, pp. 318–321.