

# 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 includes 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 very 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 its 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 may all introduce additional latency, typically greater than the latencies associated with MIDI's physical layer, and almost always less consistent. 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 it is 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 a rhythm quantizer that could transform short segments of performed notes into "appropriate" rhythmic notation in real time. One thing that sets our task apart is that we want to develop a technology that can customize its mapping so as to "best notate" the rhythms in a musician-specific way. 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 loading the processor as heavily as possible. 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 a 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 re-

stricted. 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.<sup>1</sup> 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 can be found online.<sup>2</sup>

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 there is real-time background accompaniment. Load tests do the same thing under extensive CPU load, a likely scenario when generating interactive accompaniment on-the-fly. Because our test platform is also based on *PortMidi*, a free, light-weight, cross-platform MIDI library,<sup>3</sup> users who run our tests can easily migrate from testing to writing their own PortMidi-based applications.

We have used the methodology we describe to conduct a survey and analysis of the performance of several popular MIDI interfaces on the three major consumer operating systems (Linux, Mac OS X, and Windows). While these tests are certainly not exhaustive, we believe they aptly illustrate the many issues involved in quantifying and analyzing real-time performance (as well as serving as useful points of reference in their own right). A brief overview of the performance results themselves has already been published (Nelson and Thom 2004). Our purpose in the present paper is to extend this initial overview by provide a detailed discussion of the benefits and pitfalls of various testing methodologies, analysis techniques, and statistical measures. The hope is that this

<sup>1</sup>Details from Jim Wright, in personal communication.

<sup>2</sup><http://www.cs.hmc.edu/~bthom/downloads/midi/>

<sup>3</sup><http://www.cs.cmu.edu/~music/portmusic/>

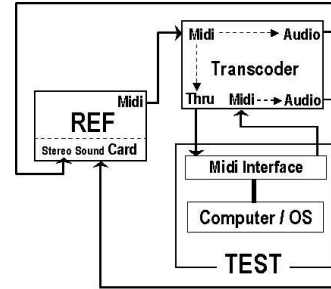


Figure 1: Overview of the system test setup.

discussion will foster wider community participation in the development of performance testing metrics. We further hope our software package and such discussion will help lead to the replacement of ad hoc performance guesses and tweaks with easy and commonplace benchmarking which system-builders may use to rigorously quantify and tune their systems’ performance.

## 2 Methodology

Our empirical testing extends the MIDI-Wave transcoder methodology proposed by Wright and Brandt (2001, 1999) by 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” (*REF*) system externally generates a reference stream of MIDI messages to send to the system whose performance is being tested (*TEST*). The *TEST* system, running a simple PortMidi application, sends them back out as the *TEST* stream. The transcoder sits between the two systems, transcoding a copy of each of the *REF* and *TEST* streams into audio signals and sending each to a different channel of the *REF* system’s soundcard 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 audio is a raw MIDI signal converted into the right voltage range for audio. This allows us to use the soundcard as a cheap and readily-available two-channel voltage sampler. When recorded, the audio sounds a bit like a buzzing fly. Some sample audio is shown in Figure 2.

As Wright and Brandt observe, recording the 31.25 kHz MIDI signal at the standard 44.1 kHz audio sampling rate suffices. Although digitally sampling the MIDI signal at this rate does introduce distortion, it is of little consequence to us—we are merely interested in locating the positions of MIDI messages. The 31.25 kHz MIDI data rate means each bit in a

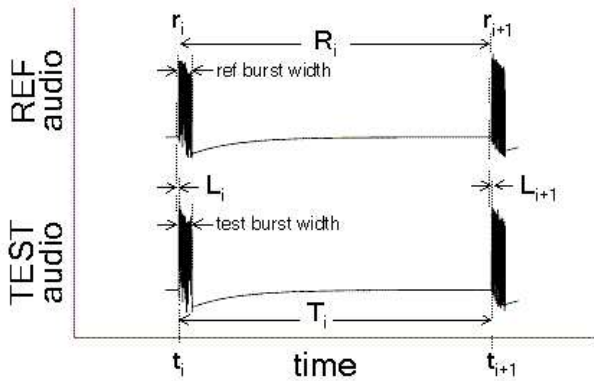


Figure 2: Sample transcoder-produced audio (from a G4 OSX 2x2 burst test).

MIDI message will remain on for approximately  $31 \mu\text{s}$ ; since sampling at a 44.1 kHz sampling rate means sampling approximately every  $23 \mu\text{s}$ , each MIDI bit will be reflected in at least a one-sample spike, so none will be missed. As detailed elsewhere (Wright and Brandt 1999), latency measurements accurate to within 0.1 to 0.2 ms are easily obtained.

## 2.2 Modifications and Proposed Benchmarks

The original Midi-Wave tests performed MIDI-thru on the TEST system using the proprietary Cubase sequencing software (whose MIDI-thru facility runs at a low system level).<sup>4</sup> We were more interested in testing under conditions similar to those an application-level software might encounter. To do so, we instead wrote a light-weight PortMidi application that simply checks for MIDI input from the REF stream once per millisecond and forwards it on to the TEST stream.

We have significantly extended the sorts of tests that can be run. Wright and Brandt’s original tests used single short, one-byte active sensing messages at fixed intervals. Though active sense messages provide a good baseline for system performance, especially since they are not typically treated in special ways by drivers or operating systems, it is more realistic patterns of MIDI traffic that we’re interested in. Therefore, we retained the ability to run active sense tests, but added a number of other ways for users of our benchmarking software to vary MIDI traffic patterns. For example, note-on and note-off messages can be used; 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 the 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.

<sup>4</sup>This and many details in this section from Jim Wright, in personal communication.

From the 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 or note-off messages (alternating) every 100 ms.
- *load burst*: same as *burst*, but with load on the 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 a quick indication, 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 10-hour test we ran.

## 2.3 Real-Time Analysis

An 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. In particular, 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.) 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 we have seen maximum latencies on the order of 25 ms. For this reason, our sense tests use a 35 ms period. Although our simple threshold scheme is more restrictive, it allows for tests of arbitrary duration, which we feel is a reasonable tradeoff. Another benefit of our algorithm is that analysis errors are unlikely to occur. Our software requires that each REF and TEST group match, and that no detected latency be larger than the period separating groups, so an error in detecting a threshold will almost certainly produce a failed test as opposed to faulty data.

The real-time audio analysis is implemented using the

cross-platform PortAudio toolkit<sup>5</sup> and is integrated with the REF stream generation, which is done using PortMidi.

## 2.4 System Configurations Tested

As listed below, we tested a selection of systems made up from commonly-used components. For brevity, we will use the italicized abbreviations 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 soundcard 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.<sup>6</sup>
- **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. Under Windows, MIDI was handled by a multimedia thread, with system priorities set as recommended on the PortAudio website: “under the *System Properties, Performance Options* menu, select ‘optimize performance for background services.’” Under Linux, the MIDI-handling thread ran with `nice` value -19.<sup>7</sup> Under OS X, the MIDI-handling thread ran as a fixed-priority (non-time-sharing) thread with precedence 30. Under all operating systems, the load thread (in tests that included load) ran as a standard thread with 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.

<sup>5</sup>PortAudio (Bencina and Burk 2001) performs a similar function for audio a PortMidi does for MIDI, and is freely available from <http://www.portaudio.com>.

<sup>6</sup>Robert M. Love’s variable-Hz (Hz=1000) and pre-emptible kernel patches and Andrew Morton’s low-latency patch.

<sup>7</sup>The software must be run as root for this heightened priority.

Previous tests (Wright and Brandt 2001) have 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 have tested both types of interfaces. We did not test the newest interface class—FireWire—as the steep prices of these interfaces (over US\$500 as of this writing) make them less ubiquitous.

Not all interfaces could be tested on all operating systems. OSX’s CoreMIDI only supports the USB interfaces, so we did not test the 4140 and SBLive there. Similarly, no Linux drivers are available for the 4140. We made numerous attempts to get the Motu to work under 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 the OSX and Windows tests, we used the newest drivers available as of November 2003 on their manufacturers’ websites. Since none of the manufacturers provide Linux drivers, we used reverse-engineered open-source drivers for these tests.<sup>8</sup> Finally, under OSX the Motu refused to forward active sense messages, so note-on and off messages were used in that case.

## 3 Terminology and Statistics

In prior work, system-induced MIDI latency has typically been characterized by latency and jitter, where *latency* is the delay introduced by a system when transmitting a MIDI message and *jitter* is how much this delay varies over time. For example, Figure 2 shows data for two bursts of MIDI (bursts  $i$  and  $i + 1$ ). REF burst start times are  $r_i$  and  $r_{i+1}$ ; corresponding TEST times are  $t_i$  and  $t_{i+1}$ . Two latency measurements can be calculated from this data:  $L_i = t_i - r_i$  and  $L_{i+1} = t_{i+1} - r_{i+1}$ . A latency measurement is obtained for each such message group, and recorded in the *Transcoder Latency* histogram (see Figure 4 for an example).

It is easy to ambiguously use latency and jitter terminology because both intimately depend on this histogram of delays. For instance, although each count in the histogram is a *specific* event’s latency, often latency is used in the context of an aggregate value: average latency (Wright and Brandt 2001), worst-case latency (Brandt and Dannenberg 1998), and so on. Fortunately, everyone seems to agree that the *distribution* that describes a system’s delay is the primary quantity of interest, and at some point in a discussion on real-time performance, jitter usually ends up being used synonymously with this distribution.

Using a different definition of latency, Brandt and Dannenberg (1998) collected “latency” measurements by calculating the difference between the actual time that separated

<sup>8</sup>The `emu10k1` ALSA driver for the SBLive, and the `usb-midi` driver for the 2x2.

adjacent timer callbacks and the constant period at which the timer was supposed to run. Because there was no external source in this case, “delay” was assumed to be zero, the assumption being that the callback itself executed very quickly (within a few microseconds). In this scheme, variation in system performance is thus the sole measurement being considered. Transcoder data not only provides latency data ( $L_i$ ), an absolute standard against which to measure delay, but period-based measurements as well; in particular REF period  $R_i = r_{i+1} - r_i$  and TEST period  $T_i = t_{i+1} - t_i$ .

Both the transcoder’s worst-case and average absolute latency estimates are useful when quantifying how responsive a system is. Period-based quantities also have merit, especially when the goal is to maintain as constant a stream of periodic inter-onset intervals as possible. For this reason, we also collect a *Transcoder  $\delta$  Latency* histogram (see Figure 4), which records differences in adjacent latencies:  $\delta L_i = L_{i+1} - L_i$ . Observe that, depending on how delay is correlated over time, the distribution over  $\delta L_i$  might look less disperse than the one over  $L_i$ .

From Figure 2 it is clear<sup>9</sup> that:

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

If the distribution of  $L_i$  were normal, with standard deviation  $\sigma$ , then the the uncertainty in  $\delta L_i$  would increase<sup>10</sup> to  $\sqrt{2} \cdot \sigma$ . In this case, we would expect more variation in the differences between adjacent latencies, which makes sense given that the addition of two random identically distributed variables only increases uncertainty. As already pointed out, however, if delays have temporal dependence, it is possible for the distribution of  $\delta L_i$  to be less spread out than the one for  $L_i$ .

Combining the definition of  $\delta L_i$  with equation 1 demonstrates its importance:

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

Therefore, in applications where we are most interested in accurately responding to a signal’s inter-onset content, the Transcoder  $\delta$  Latency histogram better quantifies how much variation a system will produce. As a more pessimistic upper-bound, *peak jitter*—the difference between the maximum and minimum latency values—quantifies the maximum possible distortion in the interval between *any* pair of events (not only adjacent events). When quickly responding to onsets themselves, the *Transcoder Latency* histogram is the primary metric of interest, and a worst-case upper-bound is provided by the maximum latency value.

To quantify bursty real-time behavior, we record the *widths* of the audio transcoded for a burst of messages. Width, as

<sup>9</sup>The widths of  $L_i$  and  $L_{i+1}$  are very close together so this may be difficult to see; a zoom on this figure would reveal that  $r_i < t_i$  and  $r_{i+1} < t_{i+1}$ .

<sup>10</sup>Propagating error by adding in quadrature.



Figure 3: A REF burst (top) is “stretched” as the TEST system is unable to keep up with the continuous burst of data (bottom).

shown in Figure 2, is the time between the beginning and end of a burst message. A distribution over width quantifies a TEST system’s capability 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 of the test system’s (bottom audio signal) inability to keep up with sending out the signal as it comes in. A thresholding scheme that handles most such cases is used to detect width, though as mentioned in Section 2.3 this relies on the bursts being well-separated.<sup>11</sup>

## 4 Results

A single test run produces a set of histograms like those shown in Figure 4. We chose to focus on this 4140 configuration because its poor performance makes for interesting discussion. Together, the Transcoder Latency and Test Width histograms provide a reasonable characterization of a system’s performance and the Transcoder  $\delta$  Latency histogram summarizes some aspects of temporal dependence. Notice the much smaller values in the  $\delta$  Latency histogram, indicating a high degree of temporal dependence (in fact, while there are many absolute latencies around 20 ms, only two  $\delta$  latencies are above the 4 ms range).

The TEST Periodic Timer histogram displays data collected by the TEST system entirely in software. This histogram displays how much variability the TEST system witnessed in its periodic scheduling of the 1-ms timer which services MIDI-thru. Rather than being measured by an independent device, this histogram’s data was measured solely with respect to the TEST system’s internal clock.

It makes perfect sense that the periodic timer and transcoder histograms correlate somewhat, for obvious possible sources of MIDI latency include an operating system’s ability to schedule things on time in the face of numerous competing requests. Having said this, one might conclude that a purely software-based approach to performance testing would suffice; indeed, at least one previous performance analysis relied on this method (Brandt and Dannenberg 1998). However, as Figure 4 illustrates, the software histogram provides a

<sup>11</sup>To help debug in cases where they are not, the software dumps an audio file containing data recorded over the past few groups, illustrating errors like the one in Figure 3.

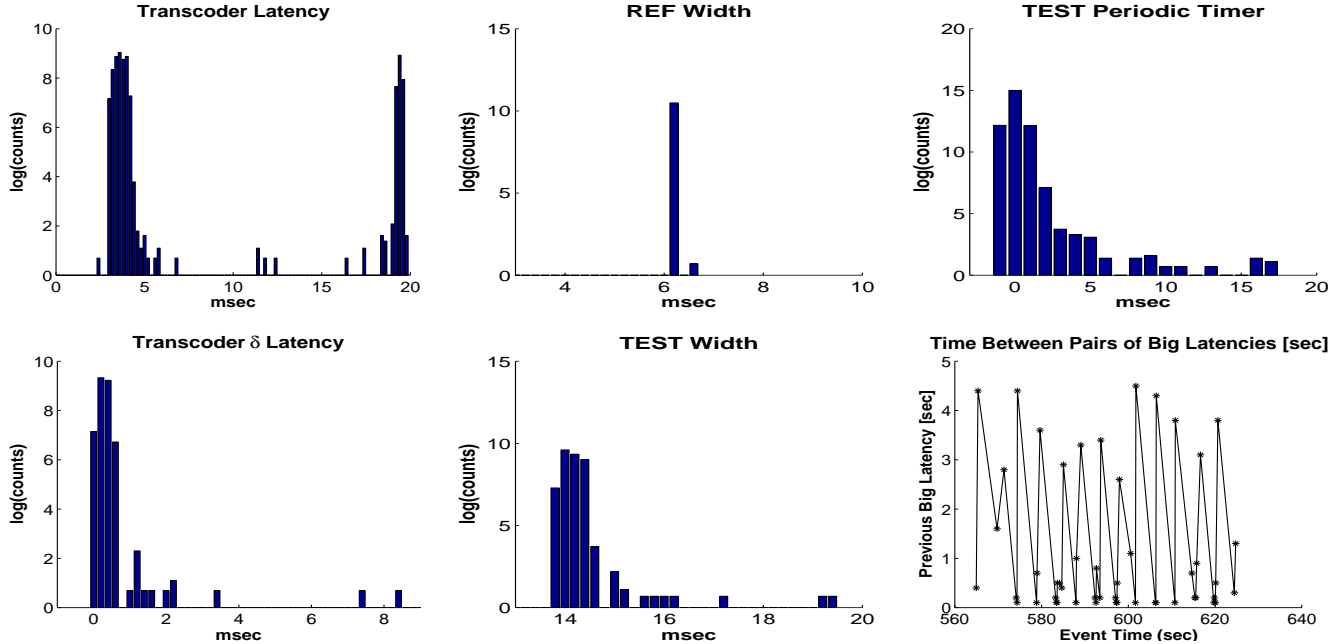


Figure 4: A sample selection of histograms (load burst on T23, Win2k, 4140), except for the lower-right-hand figure, which displays data taken from a load burst test on G4, OSX, Motu (see text for details).

much less accurate indication of the latency distribution than is available with the transcoder. For this reason, we recommend spending the extra effort needed to build the transcoder.

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 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, which suggests that MIDI input is inherently more difficult to process in real-time. For this reason, we recommend that bi-directional communication be a primary focus when measuring real-time performance.

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 here. For unloaded burst test results, see our NIME publication (Nelson and Thom 2004). Although the summary statistics in this table are useful, they do obscure valuable information about the underlying distributions. For example, since the Transcoder Latency histogram in Figure 4 is clearly bimodal, commonly used Gaussian mean and standard deviation parameters do not adequately characterize this distribution.

Histograms are not perfect either: Though they retain information about the distribution of latencies, they throw away information about how these latencies might vary over time. Some of this variation is captured by the Transcoder  $\delta$  La-

tency histogram, but this too only takes into account adjacent latencies, so does not capture many sorts of periodic behavior. The lower-right-hand plot of Figure 4 sheds additional light on temporal dependence by showing where problematic latencies occur in time. In this graph, each data point corresponds to a time (x axis) where the system had a latency greater than or equal to 7 ms. The y axis is merely the distance on the x axis between the current data point and its predecessor. In other words, data that lies close to the x axis corresponds to bursts of nearby high latencies. We have seen case where, when a system gets behind, it starts “playing catch-up”, accelerating its output of delayed events, which would explain the data-points near the x axis. Interestingly, many other problems are separated by approximately 3-5 seconds. In our cursory attempts to look more carefully at how latencies are distributed over time, we have come to realize that it would be very difficult to predict when exactly problems are likely to occur. Loosely speaking, however, the data displays some systematic temporal trends. Temporal trends, even though hard to predict, are definitely present. Temporal dependence is substantiated by Table 1; notice that the  $\delta$  latencies tend to be smaller than the absolute latencies. Further substantiation is found in the fact that the standard deviation of the  $\delta$  latencies,  $\sigma_{\delta L}$ , as opposed to being  $\sqrt{2} \cdot \sigma_L$ , is instead significantly less than  $\sigma_L$ . The deeper performance-related temporal issues certainly warrant further investigation. It has been our experience that simple modifications to our tools support fairly open-ended exploration in this area.

The good news for interactive MIDI applications is that

System	Sense msec							Load Burst msec								
	$\mu_L$	$\sigma_L$	$p_L$	$m_L$	$\mu_{\delta L}$	$\sigma_{\delta L}$	$m_{\delta L}$	$\mu_L$	$\sigma_L$	$p_L$	$m_L$	$\mu_{\delta L}$	$\sigma_{\delta L}$	$m_{\delta L}$	$p_w$	$m_w$
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 derived from histograms. The Transcoder Latency histograms are characterized by mean ( $\mu_L$ ), standard deviation ( $\sigma_L$ ), peak jitter ( $p_L$ ) and maximum ( $m_L$ ). The Transcoder  $\delta$  Latency histograms are characterized by the same set of statistics, except that peak jitter is omitted, since the minimum  $\delta L$  is zero in all cases, so peak jitter of  $\delta L$  is the same as its maximum. For the load burst tests, width is characterized by peak jitter ( $p_w$ ) and maximum width ( $m_w$ ).

the best-performing systems in our tests exhibit performance very close to the targets of 10-ms latency and 1- to 1.5-ms jitter that were 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 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. Unfortunately, although single notes have low latency and jitter, the notes towards the end of a burst have significantly higher jitter. Perceptually the impact of this behavior might lead to chords that sound slightly arpeggiated. The worst victim of system load is the 4140, which, while it outperforms the USB interfaces on a lightly-loaded system,<sup>12</sup> 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.

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 effort made by the kernel developers in addressing that

criticism has 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). Unfortunately, we have not found a good solution for PC laptops, which do not support PCI soundcards 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.

## 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—would be nice to do away with. It has been suggested to us<sup>13</sup> 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.

Also worth exploring are the “scheduled output” MIDI APIs found on many operating systems (e.g. the WinMME stream interface, which PortMidi supports). This technology

<sup>12</sup>Including in the unloaded burst tests not reported here.

<sup>13</sup>Roger Dannenberg, personal communication.

allows a MIDI message to be scheduled for output at some point in the future, instead of being sent out immediately. If messages were scheduled to be output in, say, 1 to 5 ms, this might pass off high-priority scheduling into the operating system kernel, where it may be more likely to be serviced consistently. If implemented well, this might trade off an increase in latency for a decrease in jitter, which would be desirable in some applications.

## 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<sup>14</sup> 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.

**Acknowledgments** We would like to thank a number of people for their valuable contributions: Jim Wright, Eli Brandt, and Roger Dannenberg for guidance in MIDI performance testing; Roger Dannenberg for extensive PortMidi discussion; Carl Baumgaertner and David Harris for use of their labs and circuit-building expertise; Melissa O’Neill, Claire Connelly, and Geoff Kuenning for their advice on numerous Mac and Linux systems issues; and the Harvey Mudd Computer Science Department and Faculty Research program for provid-

ing funding for and a supportive environment in which to do this research.

## 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.

<sup>14</sup><http://www.openmuse.org>