A Survey of Real-Time MIDI Performance

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

ABSTRACT
Although MIDI is often used for computer-based interactive music applications, its real-time performance is rarely quantified, despite concerns about whether it is capable of adequate performance in realistic settings. We extend existing proposals for MIDI performance benchmarking so they are useful in realistic interactive scenarios, including those with heavy MIDI traffic and CPU load. We have produced a cross-platform freely-available testing suite that is easy to use, and have used it to survey the interactive performance of several commonly-used computer/MIDI setups. We describe the suite, summarize the results of our performance survey, and detail the benefits of this testing methodology.

1. INTRODUCTION
MIDI—the most widely-used standard for interconnecting electronic music devices—was originally designed to provide low-latency transmission of messages between devices, but whether it succeeds in doing so in highly interactive real-time settings has been questioned [7, 5]. Quantifying its latency is crucial because even very small timing variations can be musically perceptible: Researchers have proposed values as low as 1 to 1.5 milliseconds as an acceptable range of latency variation [5, 7], and around 10 milliseconds as an acceptable upper bound on absolute latency [7, 3].

MIDI's fixed data rate means delays between stand-alone synthesizers sending messages of fixed size are trivial to calculate, consistent, and relatively small. Our concern is with latencies that arise when MIDI is used to communicate with software running on a general-purpose computer system, a common scenario in interactive computer music systems (e.g. [2, 4]). We use *system* to refer to a computer and all its relevant interconnected parts: MIDI interface, peripherals bus, operating system, drivers, configuration, and so on. These parts may all introduce additional latency, typically greater than the latencies of MIDI's physical layer, and certainly less consistent.

Benchmarking MIDI's latency in realistic settings is valuable, both to evaluate when its real-time performance is acceptable, and to determine the expected errors for timestamped data. Unfortunately, it rarely done, and when empirical measurements are made at all, they are often limited. For example, Wright and Brandt [9, 8] provide a method for measuring MIDI latency that is notably independent, with measurements made *externally*, rather than by the system under test. These tests, however, used single active sense messages, no system load, and had low-level system software generating responses. 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 [3], but their measurements were not external. To address these shortcomings, we have developed a freely available cross-platform software package[1] that, when used in conjunction with the inexpensive and easy-to-build *MIDI-Wave transcoder* circuit proposed by Wright and Brandt, can independently test the performance of a system in-place under a variety of realistic conditions. Such a tool is important because performance depends on so many factors that testing performance on a particular setup is desirable.

To allow our test suite to capture realistic conditions, we significantly extend upon the methodologies of Brandt, Dannenberg, and Wright. For example, in addition to repeating simpler tests, we propose more realistic *burst* and *load* tests, and these produce markedly different results. Burst tests periodically transmit groups of multiple MIDI note messages, mimicking situations that arise when there is real-time background accompaniment. Load tests do the same under CPU load, a likely scenario, especially with artificial intelligence techniques. Because our test platform is based on PortMidi, a light-weight cross-platform MIDI library,[2] it tests a configuration in which actual applications can be written.

We have used our software package, proposed benchmarks, and the MIDI-Wave transcoder 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 not exhaustive, they provide useful points of reference, and to our knowledge there is no such overview currently available. We do caution that these results should not be taken as a definitive statement on which operating systems or interfaces perform best, for performance is so system-specific. An important byproduct of this work is that it will allow others to test their own systems.

2. METHODOLOGY
We use the MIDI-Wave transcoder, as shown in Figure 1. In a given test, there are two systems: the *REF* system generates a *REF stream* of MIDI messages to send to the *TEST* system, which forwards them back out as the *TEST stream*. Copies of each stream are transcoded into audio signals and sent, one in each channel, to the REF system's soundcard line-in.[3] An example of this two-channel signal is shown in Figure 1. Delays between the two channels are analyzed in real time, as described in Section 2.2.

We modified Wright and Brandt's original tests to make them more general. A simple PortMidi application checks for MIDI input once per millisecond on the TEST system

[1] `http://www.cs.hmc.edu/~bthom/downloads/midi/`
[2] `http://www.cs.cmu.edu/~music/portmusic/`
[3] The audio signal is the raw MIDI signal converted into the right voltage range for audio, allowing us to use a soundcard as a cheap, readily-available two-channel voltage sampler.

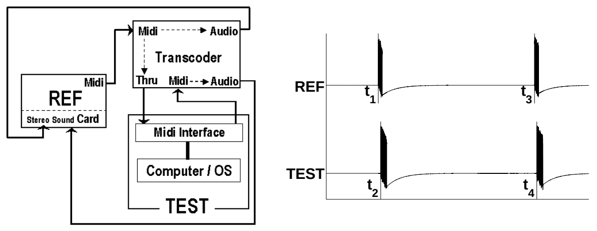

Figure 1: An overview of the MIDI-Wave setup (left) and sample transcoder-produced audio (right).

and forwards it back out; the original tests instead performed MIDI-through using the proprietary *Cubase* sequencing software. Although Cubase's MIDI-through sits at a low system level and so provides better performance, it does not allow for user software to access the incoming MIDI data, a severe limitation.

In addition to test *duration*, users can vary the *number* and *types* of messages in each burst, the *frequency* with which bursts are generated, and may also run the tests under simulated load (arbitrary arithmetic on a 1-megabyte matrix). Wright and Brandt's tests sent periodic active sensing messages, because these are typically not treated in special ways by drivers. While our software can test active sensing, we are more interested in testing performance on real notes, and especially groups of notes.

We chose three types of hour-long tests for our benchmarks:

- ***sense***: one active sensing message every 35 ms.
- ***burst***: bursts of 10 note-on/off messages every 100 ms.
- ***load burst***: same as ***burst***, but with load.

The REF computer runs an integrated cross-platform PortMidi/PortAudio[4] application. The PortMidi component generates the REF stream and the PortAudio component performs the real-time audio analysis.

2.1 Terminology

Latency is the delay produced by the system when transmitting a MIDI message; in the sample transcoder audio in Figure 1, the two latencies are $t_2 - t_1$ and $t_4 - t_3$. *Jitter* is how much this delay varies, and *peak jitter* is the difference between the maximum and minimum latencies. The importance of latency and jitter depends on the application. For interactive systems, low latency is necessary to keep systems from seeming sluggish. If latency is low, jitter—the deviation in latency—must necessarily be low. However, jitter may still be very important: In non-interactive systems, higher latency may not be as detrimental as long as jitter is low, and even in interactive systems, perceptual constraints on jitter are tighter than those on latency.

Width is the time it takes for a burst of messages to be transmitted. Our primary interest in width is that it allows us to detect when a TEST system is incapable of sending out all of the MIDI messages received in a burst in a timely manner. For instance, we've often seen spaces inserted between parts of a burst message, resulting in an overall "stretched" width. To quantify this inconsistency, we calculate the peak jitter in widths.

2.2 Real-Time Analysis

We use a simple thresholding algorithm to locate bursts in each stream. Bursts in the REF stream are matched up with corresponding bursts in the TEST stream and compared to calculate latency and width, both of which are stored in histograms. The histograms, which capture the distributional aspects of latency and width, in turn quantify jitter.

Our thresholding method requires that message "groups"—either single active sense messages or bursts of multiple messages—be well-separated, allowing us to identify 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 data collected from active sensing messages sent every 4 ms, even though they reported latencies of up to 10 ms.

Although our thresholding scheme requires larger periods, there are benefits. Light-weight real-time analysis is needed if one wants to run tests for lengthy periods of time, which can prove useful. For example, in our tests worst-case performance over an hour was in some cases 5–7 ms worse than over 15 seconds. In some cases even longer tests may be desirable, and for those the alternative to real-time analysis—storing and analyzing 6 GB of data for a 10-hour test—is not very appealing. Another benefit of our simple algorithm is that analysis errors are essentially eliminated: because our software requires that each REF and TEST group match, an error in detecting a threshold is very likely to produce a failed test rather than faulty data.

2.3 System Configurations Tested

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 (MPU-401 compatible)

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.[5]
- **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.

Previous tests [9] suggest that USB interfaces, which are newer but quickly becoming the de facto standard, perform worse than "legacy" interfaces (parallel port, PCI, serial port), so we have tested both types. We did not test

[4]PortAudio [1] performs a similar function for audio that PortMidi does for MIDI (`http://www.portaudio.com`).

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

FireWire interfaces since their expense prohibits general usage. For the OSX and Windows tests, we used the newest drivers available as of November 2003 from their manufacturers. None of the manufacturers provide Linux drivers, so reverse-engineered open-source drivers were used.[6]

Not all interfaces could be tested on all operating systems: OSX's CoreMIDI only supports the USB interfaces; no Linux drivers are available for the 4140; we were unsuccessful in getting the Motu to work under Linux; and the early revisions of Linux 2.6 available at the time of testing had USB problems on some hardware.

3. RESULTS

A single test run produces a set of histograms like those shown in Figure 2. The *Transcoder Latency* and *Test Width* histograms together reasonably characterize performance.

The *Test Callback Latency* histogram displays data collected by the TEST system in software, indicating the variability the TEST system's clock witnessed in its periodic scheduling of a 1-ms timer (the timer which serviced MIDI-thru). It makes perfect sense that the callback and transcoder histograms correlate somewhat, for an obvious possible source of MIDI latency is an operating system's ability to schedule things on time. One might be tempted to conclude that a software-based approach to performance testing would suffice; indeed, at least one previous performance analysis relied on this method [3]. However, as Figure 2 illustrates, the software histogram provides a 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 one.

Notice 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 do so very consistently. The TEST system, which had to process asynchronous MIDI input and then send it back out, had a much more difficult time. We saw this kind of behavior on virtually every system, so we recommend that bi-directional communication be a primary focus when measuring real-time performance.

Our performance survey results are summarized in Table 1 using a variety of summary statistics: the mean latency, its standard deviation, and so on. Although summary statistics are commonly used to quantify results, they tend to obscure valuable information about the underlying distributions. For example, since the leftmost histogram in Figure 2 is clearly bimodal, mean and standard deviation parameters are inadequate. Histograms are not perfect either: though they retain information about the distribution of latencies, they throw away information about how these latencies vary over time. To provide a broad overview of performance we use the summary statistics here, and discuss temporal dependence in more detail elsewhere [6]. Keep in mind that, to the degree that deviations in inter-event intervals vary somewhat systematically over time, both summary and histogram statistics might paint a more pessimistic picture than the capabilities of human perception—which occurs in time—might warrant.

The good news is that the best-performing systems in our tests exhibit performance very close to targets that researchers have proposed: 10-ms latency and 1- to 1.5-ms jitter. Our best overall performer was the SBLive on the HP desktop running WinXP. Its worst case (the load burst test) resulted in a maximum latency of 2.8 ms, peak jitter of 2.0 ms, and peak burst width jitter of 1.2 ms.

The bad news is that none of the other configurations exhibited performance at this level, at least when running under load. A common problem, exhibited by the otherwise admirably-performing 2x2 on the G4 running OSX, is fairly large width jitter under load. 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 arpeggiated. The worst victim of system load was the 4140, which, while outperforming the USB interfaces without load, degrades very badly when loaded. With minimal system load, some problems disappear, and several more interfaces have performance reasonably close to the target values. Moreover, if messages are kept relatively sparse without large bursts, about half the interfaces perform reasonably well, with jitter under 4 ms.

One pleasant result is that Linux's performance has vastly improved. For those who can tolerate 5–7-ms jitter, the SBLive on Linux 2.6 is acceptable, as are the USB interfaces on OSX. The latter will be particularly useful if G4 laptops perform similarly to desktops. Unfortunately, we have not found a good solution for PC laptops, which do not support PCI soundcards like the SBLive. While we had originally purchased the 4140, hoping that a low-level parallel port interface would perform better than the USB alternative, 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 rigorous testing.

4. FUTURE WORK

Further modifications to our testing tools are worth exploring in order to increase the range of situations that they can test. In particular, the constraint that bursts be well-separated would be nice to do away with. It has been suggested to us[7] 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. This technology allows a MIDI message to be scheduled for output at some point in the future, instead of sending it 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 might be serviced more consistently, perhaps allowing applications to trade off an increase in latency for a decrease in jitter.

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

[7]Roger Dannenberg, personal communication.

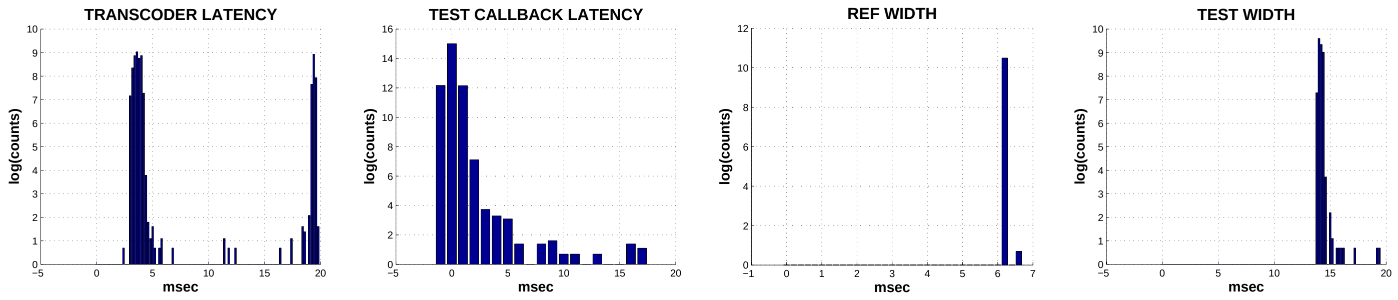

Figure 2: A sample selection of histograms (burst on T23 Win2k 4140).

System	Sense (msec)					Burst (msec)							Load Burst (msec)						
	μ	σ	m	p	≥ 10	μ	σ	m	p	≥ 10	m_w	p_w	μ	σ	m	p	≥ 10	m_w	p_w
HP Linux2.6 SB	0.8	0.3	2.3	2.1	0.0	1.1	0.3	5.5	4.9	0.0	9.6	3.4	1.2	0.3	7.6	7.0	0.0	8.6	2.4
HP Linux2.4 SB	0.8	0.4	25.6	25.4	0.6	1.2	0.4	23.5	22.9	1.0	38.8	32.6	1.2	0.4	26.6	26.0	1.1	23.9	17.7
HP Linux2.4 2x2	2.2	0.5	25.7	24.7	3.6	2.7	0.5	30.5	29.3	4.2	32.3	24.9	3.7	0.5	36.4	34.4	3.7	29.0	21.6
G4 OSX 2x2	3.5	0.4	4.6	2.2	0.0	3.6	0.4	4.8	2.4	0.0	11.6	2.2	3.6	0.4	5.8	3.2	0.0	18.1	8.7
G4 OSX Motu	5.4	0.6	7.0	3.4	0.0	5.4	0.6	8.3	4.9	0.0	10.3	8.1	5.7	0.7	9.2	5.6	0.0	10.6	7.2
HP WinXP SB	0.9	0.3	2.4	2.0	0.0	1.3	0.3	2.7	1.9	0.0	10.6	1.2	1.3	0.3	2.8	2.0	0.0	10.6	1.2
HP WinXP 2x2	3.5	0.5	5.4	3.2	0.0	5.4	0.4	7.3	4.1	0.0	12.4	3.0	5.8	0.6	7.8	5.4	0.0	12.5	3.9
HP WinXP Motu	7.5	1.5	12.2	8.0	3.1	7.8	1.5	11.6	7.0	6.5	12.9	6.5	7.9	1.5	12.6	8.0	8.3	13.2	6.8
T23 Win2k 2x2	4.3	0.6	6.3	3.9	0.0	5.6	0.6	8.4	6.0	0.0	12.6	6.0	6.8	0.5	10.6	7.8	0.0	13.6	4.2
T23 Win2k Motu	7.7	1.3	10.3	5.1	1.5	8.0	1.3	10.6	5.0	5.1	11.2	4.8	7.7	1.2	10.6	5.0	0.9	14.8	8.4
T23 Win2k 4140	2.1	0.8	4.4	3.6	0.0	4.2	0.3	5.1	2.1	0.0	16.0	2.4	3.7	0.3	20.7	18.3	25.7	19.5	5.7

Table 1: Empirical data from various system configurations. We characterize our latency histograms with the following summary statistics: the average (μ), standard deviation (σ), maximum (m), peak jitter (p), and the percentage of latencies that lies above 10 ms ($\geq$10). For the burst tests, we characterize the width histograms by the maximum burst width (m_w) and the peak width jitter (p_w).

5. CONCLUSION

Although MIDI can indeed perform close to the threshold of perceptible timing error, it is clear that performance can differ significantly as a result of system configuration, load, and MIDI traffic. Previous performance testing did not bring all of these facts to light. We hope that the survey we have presented will, in addition to illustrating some common sources of latency and jitter, encourage researchers using MIDI in interactive software settings to use independent, in-place tools to test and tune performance.

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 with this interest—existing resources such as James Wright's *OpenMuse* have similar goals. Imagine, for example, the benefits of a resource where individuals could report and discuss empirical performance measures for their particular applications and systems. Such interaction could ultimately lead to a robust and generally accepted set of useful benchmarks for interactive music applications.

Acknowledgments We would like to thank: 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 Faculty Research program and Computer Science Department for providing funding for and a supportive environment in which to do this research.

6. REFERENCES

[1] R. Bencina and P. Burk. PortAudio – an open source cross platform audio API. In *Proc. 2001 Intl. Computer Music Conf. (ICMC-01)*, 2001.

[2] J. Biles. Interactive GenJam: Integrating real-time performance with a genetic algorithm. In *Proc. 1998 Intl. Computer Music Conf. (ICMC-98)*, 1998.

[3] E. Brandt and R. Dannenberg. Low-latency music software using off-the-shelf operating systems. In *Proc. 1998 Intl. Computer Music Conf. (ICMC-98)*, pages 137–141, 1998.

[4] J. Franklin. Multi-phase learning for jazz improvisation and interaction. In *Proc. Eighth Biennial Symposium on Arts and Technology*, 2001.

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

[6] B. Thom and M. Nelson. An in-depth analysis of real-time MIDI performance. In *Proc. 2004 Intl. Computer Music Conf. (ICMC-04)*, 2004. (Submitted).

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

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

[9] J. Wright and E. Brandt. System-level MIDI performance testing. In *Proc. 2001 Intl. Computer Music Conf. (ICMC-01)*, pages 318–321, 2001.