http://cnmat.berkeley.edu/publications/multi-core-operating-system-qos-guarantees-network-audio-applications

A Multicore Operating System with QoS Guarantees for Network Audio Applications

JUAN A. COLMENARES[1,4], NILS PETERS,[1,2,3] *AES Member,* **GAGE EADS[1],**
(juan.col@samsung.com) (nils@eecs.berkeley.edu) (geads@eecs.berkeley.edu)

IAN SAXTON[1,2], ISRAEL JACQUEZ[1], JOHN D. KUBIATOWICZ[1], AND
(saxton@eecs.berkeley.edu) (mrkotfw@eecs.berkeley.edu) (kubitron@eecs.berkeley.edu)

DAVID WESSEL[1,2]
(wessel@cnmat.berkeley.edu)

[1] *Parallel Computing Laboratory, UC Berkeley, CA, USA*
[2] *Center for New Music and Audio Technologies, UC Berkeley, CA, USA*
[3] *International Computer Science Institute, Berkeley, CA, USA*
[4] *Samsung Research America – Silicon Valley, San Jose, CA, USA*

This paper is about the role of the operating system (OS) within computer nodes of network audio systems. While many efforts in the network-audio community focus on low-latency network protocols, here we highlight the importance of the OS for network audio applications. We present Tessellation, an experimental OS tailored to multicore processors. We show how specific OS features, such as guaranteed resource allocation and customizable user-level runtimes, can help ensure quality-of-service (QoS) guarantees for data transmission and audio signal processing, especially in scenarios where network bandwidth and processing resources are shared between applications. To demonstrate performance isolation and service guarantees, we benchmark Tessellation under different conditions using a resource-demanding network audio application. Our results show that Tessellation can be used to create low-latency network audio systems.

0 INTRODUCTION

Responsive real-time performance is essential to many audio applications [36]. Unfortunately, achieving this goal is very difficult or impossible with many mainstream operating systems and operating environments.

Network audio applications, for instance, can suffer from a variety of variable and unexpected delays: from potentially slow and unstable network connections to variable interrupt latencies and excessive scheduling delays. For audio applications, the entire processing chain contributes to the latency. However, many development efforts in the network audio community are directed toward network protocols with quality-of-service (QoS) guarantees (e.g., Audio Video Bridging (AVB) [19]) and low-latency compression algorithms (e.g., Fraunhofer's Ultra Low Delay (ULD) Audio Codec)—neglecting the role of the operating system (OS) at the beginning and end of the network audio chain.

Fortunately, the computer industry has recently undergone a paradigm shift away from sequential to parallel computing [7], encouraging a reexamination of the traditional software layers and the structure of the OS. Therefore, there is a unique opportunity for the audio community to raise awareness of the importance of QoS and low latency for network audio applications and eventually to contribute to the next generation of OSs.

This paper sheds a light on the role of the OS for network audio applications. We show how Tessellation OS [15,24], an experimental multicore operating system developed in the Parallel Computing Lab at UC Berkeley, can help achieve low latency and predictable behavior for real-time audio applications. Tessellation focuses on enforcing resource-allocation guarantees for client applications. By switching the focus away from utilization and toward resource management, Tessellation provides an ideal environment in which to host network audio applications.

Tessellation has been developed with an eye toward networked real-time audio applications (intended for use in live performances) [14] and other next-generation client applications (e.g., a parallel web browser [20] and a meeting diarist [17]). Not only can such applications benefit from more computational power than available from a single CPU, but they can also benefit from an execution environment with *reduced variability* and *customized scheduling*.

Tessellation has several distinctive features: (1) it provides performance isolation and strong partitioning of resources; (2) it separates global decisions about resource allocation from application-specific usage of resources (i.e., two-level scheduling); and (3) it implements an *Adaptive Resource Centric Computing* (ARCC) approach to provide adequate support for a dynamic mix of real-time, interactive, and high-throughput parallel applications. Together, these features provide a unique environment for next-generation applications.

In particular, the strong performance isolation provided by Tessellation permits software components running at the endpoints of a network audio application to yield predictable, low-jitter audio processing—providing an essential complement to the network-level improvements mentioned above. Further, the presence of user-level customized scheduling gives flexibility to exploit parallelism in an application-specific manner.

The remainder of the paper is structured as follows. The requirements of network audio applications are reviewed in Section 1. Section 2 introduces Tessellation OS and discusses some of its salient features. Then, in Section 3, we present a test network audio application, which involves a high-bandwidth, low-latency digital music interface for controlling a computationally demanding polyphonic software synthesizer. The experiments in Section 4 measure the response times of our test application running on Tessellation under different load conditions. Finally, we conclude the paper in Section 5 with our closing remarks and future work. Although we focus on audio applications in local area networks (LANs), the principles discussed here are also valid for networks covering longer distances.

1 REQUIREMENTS OF AUDIO APPLICATIONS

When performing on musical instruments, musicians usually receive acoustical feedback from their instruments within a few milliseconds (e.g., the time between hitting a drum and hearing its sound). Studies have shown that humans can perceive artificial delays of as small as 1 ms added to the acoustical feedback [30]. In multimodal virtual environments, Altinsoy [6] suggested that the tactile and auditory feedback must be within 10 ms. For a piano duet, Sawchuk et al. [32] found that the acoustical pathlength between the performers ideally should not be above 10 ms: the longer the pathlength, the harder it is to perform in synchrony, especially in fast musical sections. For most small group performances, the authors of [12] report that the maximum tolerable delay is about 50 ms. From these delay times we can establish an upper bound for the ideal end-to-end latency in audio applications.

In real-time networked audio applications, the end-to-end latency and its variability (jitter) depend on a number of accumulative factors, such as AD and DA conversions, buffering and packetization, which are part of communication protocol processing, queueing delays within the network, transmission delays, which are bounded by speed of light, and (optional) data compression and decompression. In distributed performances over large distances using wide area networks (WANs), these factors can easily accumulate to latencies much higher than the aforementioned 50 ms.

Further, audio processing stages at the sender's and receiver's sides (e.g., spatialization of audio streams [13]) can introduce additional delays depending on the completion times of the (usually) block-wise audio processes. In block-wise audio processing, a chunk of audio data is processed at once. Block-wise processing can take advantage of low-level hardware optimizations such as caching, pipelining, and single-instruction/multiple-data (SIMD) vectorization. On the other hand, the larger the block size, the longer the waiting time to buffer all incoming audio samples before the next block-wise processing can start. The choice of the right block size is consequently a tradeoff between tolerable latency and computational speed. To reduce the block size of the audio computation, the worst-case completion times of the processing tasks need to be minimized, for instance by exploiting parallel computing strategies (e.g., [11,14,34]).

In LANs the transmission delay is often low enough that it is not the dominant factor in the end-to-end latencies of networked music applications. In this case the software running on each computing node, and the operating system in particular, plays a much more significant role in meeting the latency requirements of such applications. For instance, device interrupt handling, network protocol processing, scheduling decisions, and contention on resources used by multiple applications, if not controlled properly by the OS, may result in unpredictable delays.

In the next section we present Tessellation OS and briefly discuss the salient features that allow us to control the systems software factors mentioned above in order to meet the latency requirements of LAN-based live-performance music applications.

2 OVERVIEW OF TESSELLATION OS

Tessellation [15,24] embodies an *Adaptive Resource Centric Computing* (ARCC) approach, illustrated in Fig. 1. This approach enables a simultaneous mix of interactive, real-time, and high-throughput parallel applications by automatically discovering the mix of resource assignments that maximizes the net system utility for the end user—an indicator of how well the applications are meeting their requirements and the whole system is satisfying the user's needs.

Tessellation treats resources as "first-class citizens" that can be assigned to sets of application components in order to provide predictable performance, which is essential for live-performance musical and other time-sensitive applications. Application components are grouped and execute in QoS domains called *cells*, and Tessellation distributes resources to cells. Further, each cell offers the components it hosts guaranteed access to its assigned resources. The stable, performance-isolated environment of a cell makes it possible to experimentally observe application performance metrics (e.g., completion time and throughput) and predict how these metrics vary with resources—thus enabling accurate resource optimization.

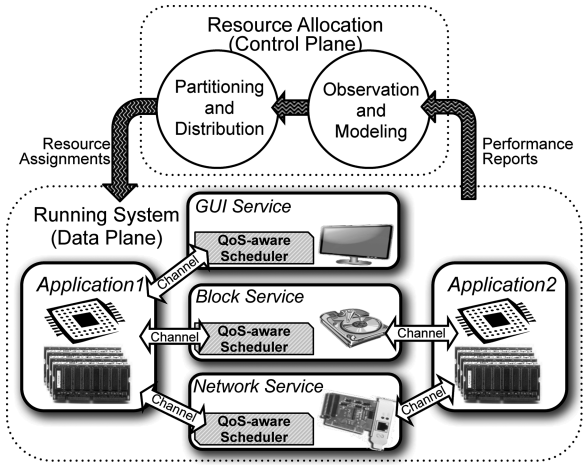


Fig. 1. Adaptive Resource-Centric Computing (ARCC): Resource allocations are automatically adjusted to maximize the overall system utility for the end user. Resources are distributed to *cells*, which provide performance isolation and guaranteed access to resources to the hosted applications and services. Cells are depicted as rounded boxes with solid lines.

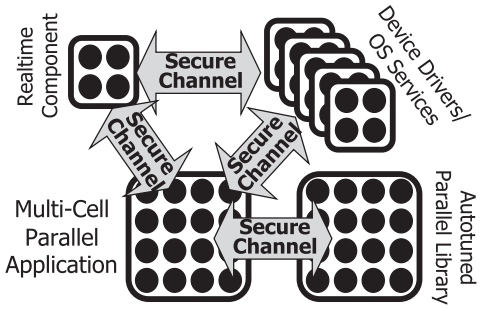


Fig. 2. Decomposing an application into a set of communicating components and services running with QoS guarantees within cells. Tessellation OS provides cells that host device drivers and OS services.

The Tessellation kernel is a thin software layer that provides support for ARCC. It implements cells and offers interfaces for cell composition and assigning resources to cells. In the rest of this section we briefly discuss the key aspects of Tessellation OS.

2.1 The Cell Model

Cells provide the basic unit of computation and protection in Tessellation. They are performance-isolated resource containers that export their resources to user level. Once resources (e.g., CPU cores and memory pages) have been assigned to cells, the Tessellation kernel gives cells full control over the usage of the resources allocated to them. Cells control their resource usage at user-level (i.e., outside the kernel) and the kernel is just minimally involved.

Applications in Tessellation are created by composing cells that communicate via efficient and secure *channels*, which enable fast asynchronous message-passing communication at user level (see Fig. 2). Channels allow an application component in a cell to access OS services (e.g., network and file services) and to interact with other application components residing in other cells.

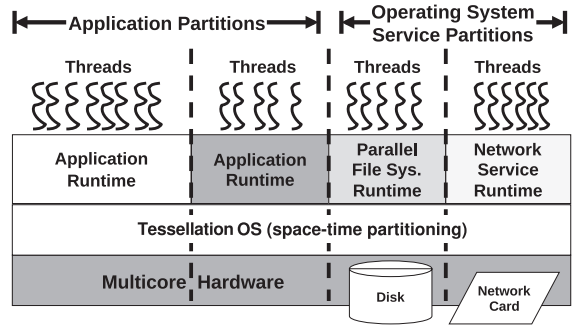


Fig. 3. Space-time partitioning (STP) in Tessellation OS: a snapshot in time with four spatial partitions.

Space-Time Partitioning [25,24]: Tessellation divides the hardware into a set of spatial partitions (see Fig. 3). Partitionable resources include CPU cores, pages in memory, and guaranteed fractional services from other cells (e.g., a throughput reservation of 150 Mbps from the network service). They may also include guaranteed cache-memory units, portions of memory bandwidth, and fractions of the energy budget, when the supporting hardware mechanisms are available (e.g., [5,22,29,31]).

Tessellation OS virtualizes spatial partitions by time-multiplexing the whole partitions onto the available hardware in a strictly controlled manner. Thus, at any given point in time some partitions can be active while others are standing by. Tessellation exports partitions to applications and OS services through the *cell abstraction*. The kernel implements a scheduling algorithm that coordinates cell switching and ensures that each cell has simultaneous access to the entire "gang" of resources assigned to its associated partition. In other words, CPU cores and other resources are *gang-scheduled* [28,16] such that cells are unaware of this multiplexing.

Tessellation provides several time-multiplexing policies for cells, some of them offering high degrees of time predictability; they are: (1) *no multiplexing* (cell given dedicated access to its assigned resources), (2) *time triggering* (cell active during predetermined and periodic time windows), (3) *event triggering* (cell activated upon event arrivals, but its contribution to the total utilization never exceeds its assigned fraction of processor time), and (4) *best effort* (cell with no time guarantees).

Two-Level Scheduling [26,15]: Two-level scheduling in Tessellation separates global decisions about resource allocation *to* cells (*first level*) from management and scheduling of resources *within* cells (*second level*). Resource redistribution occurs at a coarse time scale to amortize the decision-making cost and allow time for second-level scheduling decisions (made by each cell) to become effective.

The user-level runtimec within each cell may utilize its resources as it wishes—without interference from other cells. The cell's runtime can thus be customized for specific applications or application domains with, for instance, a particular CPU scheduling algorithm.

2.2 Service-Oriented Architecture

Cells provide a convenient abstraction for building OS services (such as network interfaces, file systems, and windowing systems) with QoS guarantees. Such services can reside in dedicated cells, have exclusive control over devices, and encapsulate user-level device drivers (see Fig. 1). Each service can thus arbitrate access to its enclosed devices and leverage the cell's performance isolation and customizable QoS-aware schedulers to offer service-time guarantees to applications and services residing in other cells.[1] Services may exploit parallelism to reduce service times or increase service throughput. Further, services can shape data and event flows coming from external sources with unpredictable behavior and prevent other cells from being affected.

Each service in Tessellation comes with a library to facilitate the development of client applications. The client libraries offer friendly, high-level application programming interfaces (APIs) to manage connections and interact with the services (i.e., they hide most of the details of inter-cell channel communication).

Two examples of services in Tessellation that offer QoS guarantees to client cells are the network service and the GUI service. The former in particular is key to low-latency networked music applications.

Network Service: This service provides access to network adapters through an API similar to the socket API [33] found in Linux and other Unix-like OSs. The network service is implemented using the lightweight TCP/IP protocol stack lwIP [2]. This service allows the specification of *minimum throughput reservations* for data flows between network adapters and client cells. It guarantees that the data flows are processed with *at least* the specified levels of throughput, provided it is feasible to do so with the networking and computational resources available to the service (e.g., the aggregate reservation should be less than or equal to the total system throughput). Moreover, the network service distributes any excess throughput proportionally among the client cells via an adaptation of the mClock algorithm [18].

GUI Service: This service provides a windowing system with response-time guarantees for visual applications [21]. It is a rearchitected version of the Nano-X Window System [3]. The GUI service exploits a user-level *earliest-deadline-first* (EDF) scheduler [23] to take advantage of multiple cores and ensure that rendering jobs with earlier deadlines are scheduled sooner. It supplements the EDF scheduler with a *resource reservation scheme*, called multiprocessor constant bandwidth server (M-CBS) [9,10], to provide different CPU reservations to different rendering tasks—a big distinction from traditional GUI systems.

Additionally, Tessellation offers a *console service* that prints character strings from other cells to a serial console (via a serial port). This service has exclusive access to the console device, and for a client cell, printing a string is just sending a one-way message on a dedicated channel to the console service. Thus, cells do not need to contend and wait for accessing the console device, and the influence of printing console messages on each application's behavior can be controlled independently and minimized. The console service was instrumental in collecting the experimental data presented in Section 5.

2.3 Adaptive Resource Allocation

Tessellation can use adaptive resource allocation to provide QoS guarantees to applications while maximizing efficiency in the system. The *Resource Allocation Policy (RAP) Service* encapsulates the decision-making logic that distributes resources to cells. It is an implementation of the upper block in Fig. 1, which involves performance monitoring (i.e., observation) and modeling as well as resource partitioning and distribution.

The RAP Service runs in its own cell and communicates with applications and other services through channels. It decides how resources should be divided among cells in the system by monitoring other cells, adapting their resources in response to changing conditions, and controlling the admission of new cells into the system.[2] The RAP Service communicates the allocation decisions to the kernel via a system call and to services through their *QoS Specification* interfaces over channels.

Note that the user can specify the set of resources to be given to a cell and indicate to the RAP Service that such resource allocation cannot change. This feature is very valuable to musical performers. It gives them complete control over the resources allocated to their musical applications. Further, it assures performers that their applications will behave as expected because they can recreate the execution conditions in which the applications were tested, profiled, and optimized.

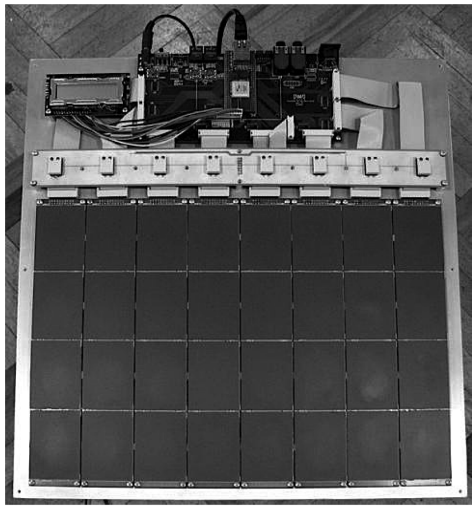
More details on adaptive resource-allocation in Tessellation appear in [15].

3 A LAN MUSICAL APPLICATION

The SLABS control interface, third-prize winner at the 2009 Guthman Instrument Design Competition [1], and the Migrator Synthesizer program constitute a real-world example of a low-latency computationally-demanding network audio application. The Migrator Synthesizer, originally created in Max/MSP [37], was redesigned and implemented for Tessellation to determine how well our OS meets the requirements of network audio applications (see Section 4). In this section we describe the components of the Migrator Synthesizer program and its implementation on Tessellation OS. Fig. 5 depicts the signal flow between components of the application.

[1] In keeping with ARCC, we view the services offered by such *service cells* as additional resources to be managed by the adaptive resource allocation architecture.

[2] The RAP Service refuses to admit new cells whose resource requirements are incompatible with existing obligations to other cells.



(a) Bird's eye view of the interface.

(b) Playing technique (side view)

Fig. 4. The SLABS multitouch interface.

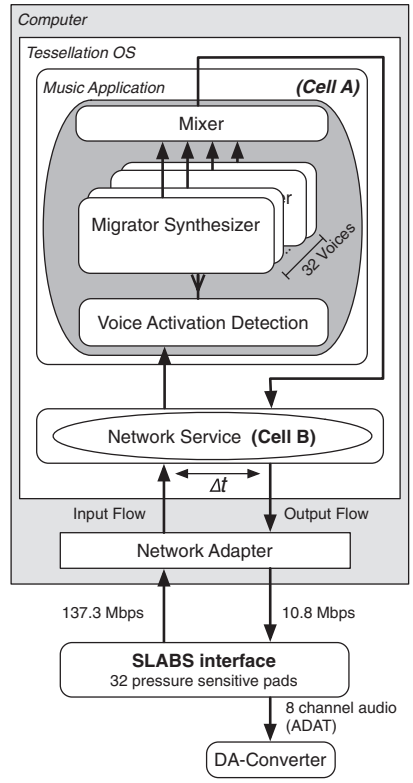


Fig. 5. LAN music application: structure and signal flow.

3.1 The SLABS Multitouch Interface

The SLABS interface [35] has 32 pressure-sensitive touchpads (see Fig. 4(a)). The horizontal and vertical position, as well as pressure of each pad are measured, calibrated, and packetized using low-latency field programmable gate arrays (FPGAs) and broadcasted as 96 audio channels at 44.1 kHz in 32-bit resolution via UDP over Ethernet. The bandwidth requirement of the resulting output data stream is 137.3 Mbps, or the bit-rate of about 100 compact discs played simultaneously.

To enable highly expressive tactile control over the Migrator Synthesizer, the position and pressure of the fingers on a SLABS pad must be accurately sampled with a high rate. For this reason the device uses digital audio streaming to transmit the gesture data because it supports a high sample rate and low temporal jitter, whereas using the MIDI protocol generally limits the sampling rate and introduces sample timing uncertainty due to the lack of timestamps.

The SLABS interface can also receive data. Eight audio channels can be sent from the host to the interface (10.8 Mbps), which are accessible via an ADAT lightpipe outlet for external DA conversion, e.g., for headphone monitoring.

Additionally, the Open Sound Control (OSC) protocol can be used to program a small display and two LEDs associated with each pad. Processes on the host computer can also be controlled via eight toggles at the top of the pads.

Network Protocol: The SLABS interface uses a custom-made network protocol [8]. It produces one UDP packet every 45.35 μs. A packet contains 2 temporal frames, each with 3 single-precision values for each of the 32 pads, corresponding to the x-position, y-position, and pressure of the finger. Each UDP packet sent back to the SLABS contains data for 32 temporal frames, each consisting of amplitude values for the 8 output audio channels.

3.2 The Migrator Synthesizer Program

The SLABS interface controls the Migrator Synthesizer program that runs under Tessellation OS (see Fig. 5). The program instantiates 32 Migrator Synthesizer objects, each one using 100 sinusoidal oscillators. The oscillators change frequency in a staggered fashion, with each one migrating every several seconds. New frequency values are sampled from a distribution that is computed by convolving a chosen microtonal pitch probability table with a gaussian function of specified standard deviation.

Each touchpad on the SLABS is used to control one of the Migrator Synthesizer objects, thus allowing independent control of up to 32 polyphonic voices. The y-position of the finger on the touchpad is used to select among four different probability tables assigned to each pad, the x-position is used to set the width of the gaussian function that adds randomness, and the pressure is used to modulate amplitude (it is silent when no force is applied).

Implementation: The Migrator Synthesizer was implemented in the FAUST environment [27] and exported as C++ code, which was then integrated into our Tessellation application program. As shown in Fig. 5, the output signals produced by the Migrator Synthesizer objects are mixed by the Mixer object and then sent to the SLABS via the network service. Also, a simple voice activation, which

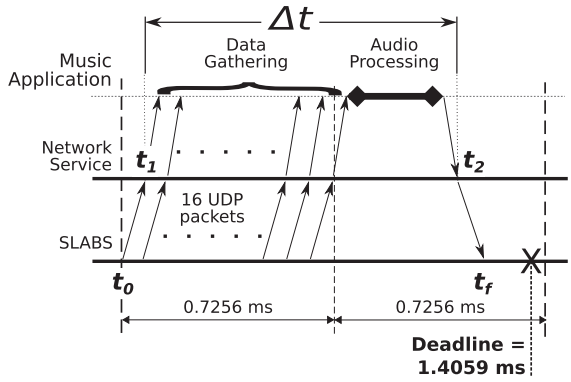


Fig. 6. Response time deadline for the SLABS and measured response time (Δt) of the music-application/network-service pair.

detects if a SLABS pad is pressed, ensures that only the Migrator Synthesizer objects associated with active pads are processed.

Within every interval of 0.7256 ms the SLABS interface sends a series of 16 UDP packets that form a complete input data package, as shown in Fig. 6. Once the 16th UDP packet has arrived, the *data gathering phase* is complete and the block-wise *audio processing phase* can start. Since the audio processing phase of the current input data package overlaps with the data gathering phase of the next input package, the audio processing of the current package needs to finish before the data gathering phase of the next package completes. Therefore, the overall time for delivering an output data package is 1.4521 ms—twice the operation period (0.7256 ms) of the SLABS. However, our deadline must be slightly smaller. The reason is that t_0 is the time at which the SLABS sends to the network a UDP packet containing two temporal frames after they have been digitized. Therefore, our deadline is 1.4059 ms ($(64-2)$ samples$/44.1kHz$).

Given this demanding low-latency requirement and considering the amount of audio data to be processed, both the Migrator Synthesizer program and the network service were deployed in non-multiplexed cells (Cells A and B in Fig. 5). Those cells were statically allocated and given dedicated access to several hardware threads[3] in the system.

To further account for the low-latency requirement, we implemented a custom-made user-level runtime for the Migrator Synthesizer program. From hardware threads allocated to the cell, the runtime dedicates one hardware thread to continuously handle communication with the network service and dynamically assigns the rest of the hardware threads for the Migrator Synthesizer objects to run.

The runtime disables preemption such that each Migrator Synthesizer object processes each input data block without jitter-inducing, periodic timer interrupts. Timer interrupts can introduce variability by executing kernel code at unpredictable points in the audio processing. Another effect of timer interrupts is that hardware units that speed up computation by caching memory from previous executions or predicting results of future ones (e.g., CPU's cache, translation lookaside buffer (TLB), and branch predictors) can become "polluted" by the infrequently executed kernel code.

[3] A hardware thread is a single physical processing engine, i.e., an entire single-threaded core or one hardware-thread context in a multi-threaded core (e.g., Intel's Core i7 with hyper-threading enabled).

4 EXPERIMENTAL EVALUATION

In this section we examine the level of performance isolation that Tessellation OS can offer to the real-time networked music application presented in the previous section. We also evaluate the ability of Tessellation's network service to provide bandwidth guarantees to the application.

We measured the *aggregate* response time of the music application and the network service under different execution conditions. The response time of the music-application/network-service pair, denoted as Δt, was measured in the network driver, as close as possible to the network adapter (see Fig. 5). The network driver is part of the network service and runs at user level. As shown in Fig. 6, Δt is the elapsed time between t_1 and t_2, where:

- t_1 is the instant at which the driver receives the interrupt from the network adapter indicating that the first UDP packet of a series of sixteen has arrived from the SLABS, and
- t_2 is the instant at which the driver writes the output packet, corresponding to the input data, to the network adapter making it send the packet to the SLABS.

The difference $t_2 - t_1$ is a close approximation to the response time observed by the SLABS at its network port (i.e., $t_f - t_0$) because the computer running Tessellation and the SLABS are the only nodes connected to the gigabit Ethernet LAN.

All measurements were taken using a controlled input; i.e., the SLABS interface was always driven with the same pressure input on a fixed set of pads. We chose an input that, when running the music application and the network service alone, could produce response-time values close to, but below the deadline of 1.4059 ms.

We used an Intel system with two 2.66-GHz Xeon X5550 quad-core processors, hyper-threading enabled (i.e., 16 hardware threads), and a 1-Gbps Intel Pro/1000 Ethernet network adapter.

4.1 Performance Isolation

In this experiment we measured the aggregate response time of the music application and the network service when the pair ran alone and together with other applications in separate cells.

Table 1 lists the applications, services, and their hosting cells used in this experiment. The cells were divided into two groups. The *Group 1* comprised the music application in Cell A and the OS services required for its operation; namely, the network service in Cell B and the console service in Cell C. The cells in Group 1 were statically allocated

Table 1. Applications and services used to evaluate performance isolation.

	Cell	Application	Cell Type	Hardware Threads
Group 1	A	Music Application	Non-multiplexed	4,5,6,7
	B	Network Service	Non-multiplexed	1,2,3
	C	Console Service	Non-multiplexed	10
Group 2	D	Video Player 1	Time-triggered	8
	E	Video Player 2	Event-triggered	9
	F	GUI Service	Non-multiplexed	11,12
	G	EP Benchmark	Best-effort	8
	H	EP Benchmark	Best-effort	9,13
	I	EP Benchmark	Best-effort	9,13,14,15

Note: The hardware thread 0 was dedicated to kernel monitoring and was not assigned to any cell.

and given dedicated access to specific hardware threads in the system. Cell A was assigned two entire CPU cores, each with two hardware threads.

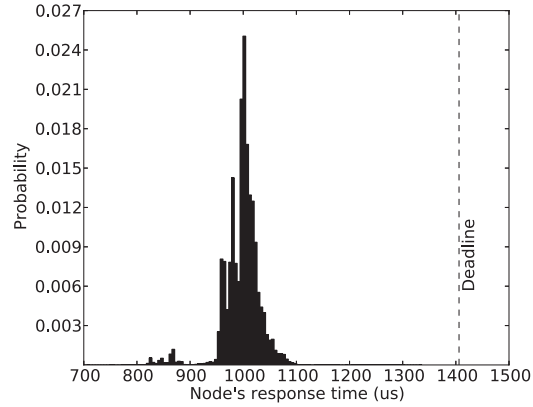
The *Group 2* included Cells D to I. These cells were added to try to interfere with the music application and influence the response times of the application codes running in Cells A and B. The cells in Group 2 used different multiplexing policies and hosted non-musical applications. Cell D, a time-triggered cell, and Cell E, an event-triggered, each contained a video player program that sent 30 frames per second to the GUI service in Cell F. Cells G, H, and I were best-effort cells with different hardware-thread counts, and they continuously ran the embarrassingly parallel (EP) kernel from the NAS Parallel Benchmarks [4].

In this experiment only the music application had access to the network service. Paging[4] was not available to any application.

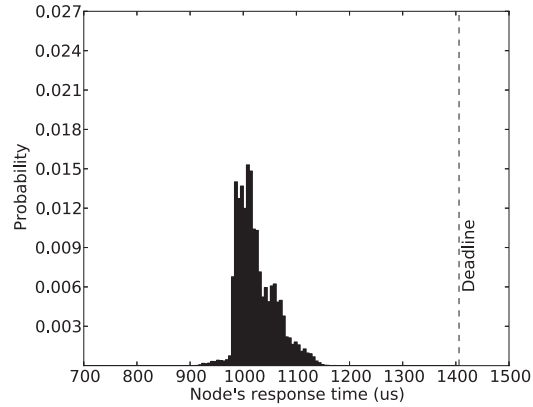
We ran the cells in Group 1 with and without the cells in Group 2 for three minutes, resulting in about 250,000 response-time measurements for each run. For the purpose of this paper we believe this amount of time is sufficient to characterize the level of performance isolation that Tessellation offers to the music application. (Incidentally, we have previously tested the system in an hour-long live demonstration for its long-term stability.) Before we started the measurements, a warm-up phase of one second was granted to the system to ensure that our measurements were unaffected by non-deterministic initialization effects related to hardware (e.g., cold caches and branch predictors) and software (e.g., creation of synthesizer objects).

Fig. 7 shows the histograms and first-order statistics obtained from the measured response times. Most important, the measurements indicate that there are no significant differences in the response times across both scenarios. The observed maximum response times are slightly below the deadline (1.4059 ms). Further, all observed response times are larger than the lower bound of 0.7256 ms (see Fig. 6) and the mean response times are around 1 ms.

A small difference between Fig. 7(a) and Fig. 7(b) is noticeable: Compared to the scenario where the cells in Group 1 were running alone, the scenario with both cell groups exhibits a slightly larger maximum response time (by 69μs) and a slightly longer latency tail. We attribute this difference to interference in the shared caches and memory interconnect particularly due to the video player cells' large, streaming video file. Despite this, the mean response time stayed within 27μs of the single-cell group experiment and deadlines were made in both experiments. Therefore, with the established hardware partitioning, Tessellation was able to provide sufficient performance isolation to both our



(a) Cells in Group 1 running alone (0 missed deadlines, $max = 1336\mu s$, $mean = 997.54\mu s$, $stdev = 35.11\mu s$).



(b) Cells in Groups 1 and 2 running together (0 missed deadlines, $max = 1405\mu s$, $mean = 1024.61\mu s$, $stdev = 36.88\mu s$).

Fig. 7. Histograms and first-order statistics for the aggregate response times of the music application and the network service.

[4] Paging is a memory-management scheme by which a computer can store and retrieve data, in blocks called *pages*, from secondary storage (e.g., hard disk drive) for use in main memory.

music application and the network service, within the limitations of our test hardware platform.

4.2 Network Service's Bandwidth Guarantees

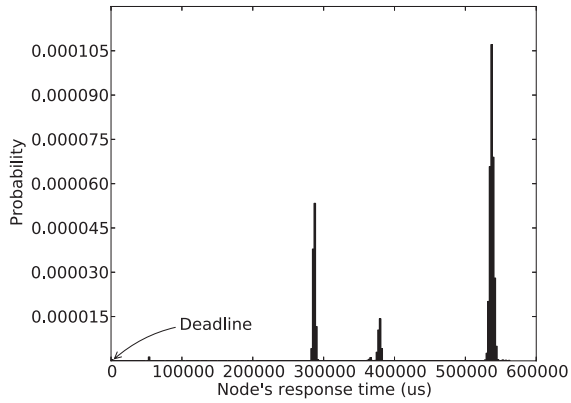
In this experiment we measured the aggregate response time of the music application and the network service when they both ran along with an *abusive* data streaming application on a separate cell. The abusive application was designed to consume as much available bandwidth as the network service allows, thereby trying to prevent the music application from using network service's resources. The abusive application was hosted in a new cell (Cell X), added to the Group 1 of our previous experiment (Section 4.1), and no cell in Group 2 was used in this experiment. Cell X was a non-multiplexed cell assigned hardware thread 14. No bandwidth reservation was given to Cell X in the network service.

We compare two scenarios. In the first scenario, the network service provides no guarantees to any application. Under the influence of the abusive application, we expected that the aggregate response time would increase. In the second scenario, the network service guarantees a bandwidth of 240 Mbps to the music application, which would make the audio processing unaffected by the abusive application.

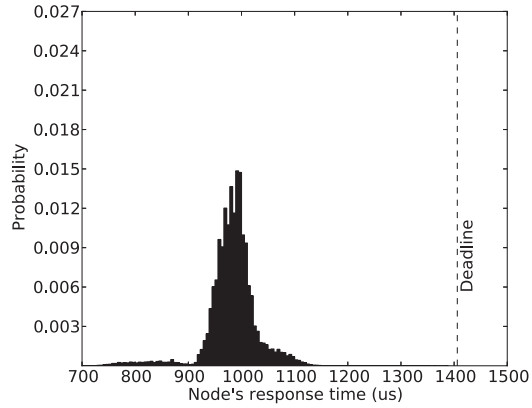
Fig. 8 shows the histograms and first-order statistics obtained from the aggregate response times in both scenarios. It is clearly visible that the aggregate response significantly differ. When the network service provided no bandwidth guarantee to the music application, the response time was severely deteriorated. The fact that 99.91% of all deadlines are missed, makes the music application unusable (see Fig. 8(a)).

On the contrary, when the network service guaranteed the specified bandwidth to the music application, the influence of the abusive streaming application was barely noticeable (see Fig. 8(b)). The music-application/network-service pair exhibited a behavior that was close to the behavior when running the music application in isolation (Fig. 7). The response-time values were now below the deadline, indicating that Tessellation's network service was able to guarantee the bandwidth demands of the music application.

We also ran the experiment with the network service providing the music application a bandwidth guarantee of 150 Mbps—a value closer to the actual application's bandwidth requirement. In this case we observed eight missed deadlines. These deadline misses happened because the network service currently guarantees bandwidth but not latency. Reserving more bandwidth than required was needed to compensate for this.



(a) Network service without bandwidth guarantees (243903 missed deadlines, $max = 562898\mu s$, $mean = 461133.61\mu s$, $stdev = 113320.00\mu s$).



(b) Network service with bandwidth guarantee to the music application (0 missed deadlines, $max = 1389\mu s$, $mean = 983.20\mu s$, $stdev = 48.37\mu s$).

Fig. 8. Histograms and first-order statistics for the aggregate response times of the music application and the network service in presence of an abusive data streaming application.

5 CONCLUDING REMARKS

This paper discussed the advantages of Tessellation OS, an experimental multicore operating system, for guaranteeing quality of service (QoS) at the edges of the network audio chain. Different experiments were conducted using a real-world, resource-demanding network music application within a LAN environment. We measured the aggregate response times of the audio DSP process and the network service within Tessellation under different load conditions. Our results show that Tessellation enables network audio applications to meet their time requirements. Tessellation's features, such as performance isolation, customizable user-level schedulers, and a network service able to guarantee minimum throughput reservations to data flows, have proved essential for network audio applications to achieve low-latency audio streaming and processing within computer nodes.

For audio applications in wide area networks (WANs), Tessellation OS can play an important role in providing end-to-end latency guarantees when complemented with suitable network protocols. For that reason, we are interested in evaluating how effective Tessellation would be for audio applications in a WAN environment. WANs (e.g., the Internet) often suffer from large and jittery transmission delays. In this case, it might even more important to have an operating system that minimizes additional delays.

As part of our future work we plan to compare the performance of the Migrator Synthesizer program on Tessellation and other operating systems (e.g., Linux). In addition, we are currently rearchitecting the network service in order to provide latency guarantees to data flows between network adapters and client cells.

6 ACKNOWLEDGMENTS

This research was supported by Microsoft (Award #024263), Intel (Award #024894), matching U.C. Discovery funding (Award #DIG07-102270), and DOE ASCR FastOS Grant #DE-FG02-08ER25849. Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung. Nils Peters is supported by the German Academic Exchange Service (DAAD). No part of this paper represents the views and opinions of the sponsors mentioned above.

We thank the other members of the Par Lab for their collaboration and feedback on the ideas presented in this paper. Finally, we want to thank the anonymous reviewers for their valuable comments and suggestions.

7 REFERENCES

[1] The 2009 winner of the Guthman Musical Instrument Competition. http://www.gatech.edu/newsroom/release.html?nid=39716. Accessed: Dec 2012.

[2] lwIP - a lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/. Accessed: Dec 2012.

[3] Nano-X window system. http://microwindows.org/. Accessed: Dec 2012.

[4] NAS parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html. Accessed: Dec 2012.

[5] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A Predictable SDRAM Memory Controller," *Proc. of the 5th IEEE/ACM Int'l Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*, pp. 251-256, Salzburg, Austria (2007).

[6] M. E. Altinsoy, "The Quality of Auditory-Tactile Virtual Environments," *J. Audio Eng. Soc.*, vol. 60, pp. 38–46 (2012 Jan./Feb.).

[7] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A View of the Parallel Computing Landscape," *Communications of the ACM*, vol. 52, no.10, pp. 56–67 (2009).

[8] R. Avizienis, A. Freed, T. Suzuki, and D. Wessel, "Scalable Connectivity Processor for Computer Music Performance Systems," *Proc. of the Int'l Computer Music Conference*, pp. 523–526, Berlin, Germany (2000).

[9] S. Baruah, J. Goosens, and G. Lipari, "Implementing Constant-Bandwidth Servers upon Multiprocessors," *Proc. of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*, pp. 154–163, San Jose, CA, USA (2002).

[10] S. Baruah and G. Lipari, "Executing Aperiodic Jobs in a Multiprocessor Constant-Bandwidth Server Implementation," *Proc. of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, pp. 109–116, Catania, Italy (2004).

[11] E. Battenberg, A. Freed, and D. Wessel, "Advances in the Parallelization of Music and Audio Applications," *Proc. of the Int'l Computer Music Conference*, New York, USA , pp. 349–352 (2010).

[12] N. Bouillot, E. Cohen, J. R. Cooperstock, A. Floros, N. Fonseca, R. Foss, M. Goodman, J. Grant, K. Gross, S. Harris, B. Harshbarger, J. Heyraud, L. Jonsson, J. Narus, M. Page, T. Snook, A. Tanaka, J. Trieger, and U. Zanghieri, "AES White Paper: Best Practices in Network Audio," *J. Audio Eng. Soc.*, vol. 57, pp. 729-741 (2009 Sept.).

[13] J. Braasch, N. Peters, and D. L. Valente, "Sharing Acoustic Spaces over Telepresence Using Virtual Microphone Control," presented at the *123rd Convention of the Audio Engineering Society (*2007), convention paper 7209.

[14] J. A. Colmenares, E. Battenberg, R. Avizienis, I. Saxton, N. Peters, K. Asanovic, J. Kubiatowicz, and D. Wessel, "Real-Time Musical Applications on an Experimental Operating System for Multi-Core Processors," *Proc. of the Int'l Computer Music Conference*, Huddersfield, England (2011).

[15] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Resource Management in the Tessellation Manycore OS," *Proc. of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, Berkeley, CA, USA (2010).

[16] D. G. Feitelson and L. Rudolph, "Gang Scheduling Performance Benefits for Fine-Grain Synchronization," *J. Parallel and Distributed Computing*, vol. 16, pp. 306–318 (1992).

[17] G. Friedland, J. Chong, and A. Janin, "A Parallel Meeting Diarist," *Proc. of the Int'l Workshop on Searching Spontaneous Conversational Speech (SSCS '10)*, pp. 57–60, Firenze, Italy (2010).

[18] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling throughput Variability for Hypervisor IO Scheduling," *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, pp. 1–7, Vancouver, BC, Canada (2010).

[19] IEEE. Audio/video bridge task group, http://www.ieee802.org/1/pages/avbridges.html. Accessed: Dec 2012.

[20] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik, "Parallelizing the Web Browser," *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, Berkeley, CA, USA (2009).

[21] A. Kim, J. A. Colmenares, H. Alkaff, and J. Kubiatowicz, "A Soft Real-Time Parallel GUI Service in Tessellation Many-core OS," *Proc. of ISCA 27th Int'l Conference on Computers and Their Applications (CATA 2012)*, Las Vegas, Nevada (2012).

[22] J. W. Lee, M. C. Ng, and K. Asanović, "Globally-Synchronized Frames for Guaranteed Quality-of-Service in On-Chip Networks," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 89–100 (2008).

[23] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46–61 (1973).

[24] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-Time Partitioning in a Manycore Client OS," *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, Berkeley, CA, USA (2009).

[25] L. Luo and M.-Y. Zhu, "Partitioning Based Operating System: A Formal Model," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 3, pp. 23–35 (2003).

[26] R. Obermaisser and B. Leiner, "Temporal and Spatial Partitioning of a Time-Triggered Operating System Based on Real-Time Linux," *Proc. of the 11th IEEE Int'l Symposium on Object Oriented Real-Time Distributed Computing (ISORC'08)*, pp. 429–435, Orlando, Florida, USA (2008).

[27] Y. Orlarey, D. Fober, and S. Letz "An Algebra for Block Diagram Languages," *Proc. of the Int'l Computer Music Conference*, pp. 542–547, Gothenburg, Sweden (2002).

[28] J. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. of the 3rd Int'l Conference on Distributed Computing Systems (ICDCS'82)*, pp. 22–30, Miami/Ft. Lauderdale, FL, USA (1982).

[29] M. Paolier, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 57–68 (2009).

[30] D. Ronken, "Monaural Detection of a Phase Difference between Clicks," *J. Acous. Soc. Am.*, vol. 47, pp. 1091–1099 (1970).

[31] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and Efficient Fine-Grain Cache Partitioning," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3, pp. 57–68 (2011).

[32] A. Sawchuk, E. Chew, R. Zimmermann, C. Papadopoulos, and C. Kyriakakis, "From Remote Media Immersion to Distributed Immersive Performance," *Proc. of the ACM SIGMM Workshop on Experiential Telepresence (ETP'03)*, pp. 110–120, Berkeley, CA, USA (2003).

[33] W. R. Stevens, B. Fenner, and A. M. Rudoff, *Unix Network Programming, Volume 1: The Sockets Networking API, 3rd Edition* (Addison-Wesley Professional, 2004).

[34] N. Tsingos, W. Jiang, and I. Williams, "Using Programmable Graphics Hardware for Acoustics and Audio Rendering," *J. Audio Eng. Soc.*, vol. 59, pp. 628–646 (2011 Sept.).

[35] D. Wessel, R. Avizienis, A. Freed, and M. Wright, "A Force Sensitive Multi-Touch Array Supporting Multiple 2-D Musical Control Structures," *Proc. of the 7th Int'l Conference on New Interfaces for Musical Expression (NIME'07)*, pp. 41–45, New York, NY, USA (2007).

[36] D. Wessel and M. Wright, "Problems and Prospects for Intimate Musical Control of Computers," *Computer Music J.*, vol. 26, no.3, pp. 11–22 (2002).

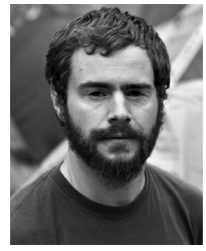
[37] D. Zicarelli, "An Extensible Real-Time Signal Processing Environment for Max," *Proc. of the Int'l Computer Music Conference*, pp. 463–466, Ann Arbor, MI, USA (1998).

THE AUTHORS


Juan A. Colmenares


Nils Peters


Gage Eads


Ian Saxton


Israel Jacquez


John Kubiatowicz


David Wessel

Juan A. Colmenares is a staff research scientist in the Computer Science Laboratory at Samsung Electronics R&D Center in San Jose, California. He is also an industrial researcher for Samsung in the Ubiquitous Swarm Lab at UC Berkeley. He completed his postdoctoral training in the Parallel Computing Laboratory (Par Lab) at UC Berkeley in 2012. He received a Ph.D. degree in computer engineering from UC Irvine in 2009. He also obtained an M.S. in applied computing in 2001 and a B.S. in electrical engineering in 1997, both from the University of Zulia, Venezuela. His research interests include operating systems for many-core architectures, real-time distributed computing, and multimedia applications.

●

Nils Peters is a postdoctoral fellow at the International Computer Science Institute (ICSI) and the Center for New Music and Audio Technologies (CNMAT) at UC Berkeley, California. He holds a M.Sc. degree in electrical and audio engineering from the University of Technology in Graz, Austria, in 2005 and a Ph.D. in music technology from McGill University in Montreal, Canada, in 2011. He has worked as an audio engineer in the fields of recording, postproduction, and live electronics and is currently working on real-time algorithms for sound-field analysis with large scale microphone arrays. He co-develops the open-source media-processing library Jamoma and organizes UC Berkeley's lecture series on spatial audio technology and room acoustics.

●

Gage Eads is a Ph.D. student in the Parallel Computing Lab (Par Lab) at UC Berkeley. He received a B.S. degree with highest honors in electrical engineering from UT Austin. His research interests include many-core I/O acceleration, memory system design, and operating systems.

●

Ian Saxton is a Ph.D. student in music composition at UC Berkeley and researcher at the Center for New Music and Audio Technologies (CNMAT) and Parallel Computing Lab (Par Lab). He received a B.A. in music with an electronic music minor from UC Santa Cruz in 2005, and a Masters in computer music from UC San Diego in 2008. His interests include parallel audio scheduling, music programming languages, live electronic music control, automatic music analysis/generation, and playing the drums.

●

Israel Jacquez received a B.S. in electrical engineering and computer science from UC Berkeley in 2012 and is currently working in a start-up company in San Francisco with an initial focus on mobile games. He was an undergrad researcher in the Parallel Computing Lab (Par Lab) at UC Berkeley. His interests include compilers and operating systems.

●

John Kubiatowicz received a double B.S. in electrical engineering and physics, 1987, M.S. in electrical engineering and computer science, 1993, and a Ph.D. in electrical engineering and computer science as well a Minor in physics, 1998, all from M.I.T. He joined the faculty of EECS at UC Berkeley in 1998. Current research interests include the design of new operating systems for multicore and many-core systems, with a particular focus on resource allocation and quality of service. Other interests include the design of extremely-wide area storage utilities (cloud storage) as well as secure protocols and routing infrastructures that provide privacy, security, and resistance to denial of service.

●

David Wessel studied mathematics and experimental psychology at the University of Illinois and received a doctorate in mathematical psychology from Stanford in 1972. His work on the perception and compositional control of timbre in the early 70s at Michigan State University led to a musical research position at IRCAM in Paris in 1976. In 1979 he began reshaping the Pedagogy Department to link the scientific and musical sectors of IRCAM. In 1985 he established a new IRCAM department devoted to the development of interactive musical software for personal computers. In 1988 he began his current position as Professor of Music at the University of California, Berkeley where he is Director of CNMAT. He is particularly interested in live-performance computer music where improvisation plays an essential role. He has collaborated in performance with a variety of improvising composers including Roscoe Mitchell, Steve Coleman, Ushio Torikai, Thomas Buckner, Vinko Globokar, Jin Hi Kim, Shafqat Ali Khan, and Laetitia Sonami has performed throughout the US and Europe.