REAL-TIME MUSICAL APPLICATIONS ON AN EXPERIMENTAL OPERATING SYSTEM FOR MULTI-CORE PROCESSORS

Juan A. Colmenares[1], *Ian Saxton*[1,2], *Eric Battenberg*[1,2], *Rimas Avižienis*[1,2], *Nils Peters*[1,2,3]
Krste Asanović[1,3], *John D. Kubiatowicz*[1], *David Wessel*[1,2]

[1]Par Lab [2]CNMAT [3]ICSI
University of California, Berkeley, USA Berkeley, CA, USA

ABSTRACT

A natural approach to increasing the performance of musical applications is to exploit their inherent parallel structure on general-purpose multi-core architectures. In this paper, we discuss opportunities for exploiting parallelism in audio DSP graphs as well as within select audio processing components. We describe *Tessellation OS*, an experimental operating system structured around resource distribution, performance isolation, and QoS guarantees, and *Lithe*, a user-level runtime framework that enables construction of composable, application-specific schedulers. We present the design and implementation of a real-time parallel musical application on top of Tessellation OS and conclude with some preliminary experimental results.

1. INTRODUCTION

The age of parallel computing is upon us [36, 3]. Power consumption has put a limit on processor clock rates, and techniques for extracting instruction-level parallelism have faltered. It appears that the only path to increased performance requires embracing multi-core architectures [1]. But the move to parallel programming is not without difficulties. It is generally acknowledged that parallel programming is hard; there are race conditions, deadlocks, and complex patterns of communication and synchronization among tasks. Many software developers have asked for automatic parallelization of programs – simply to avoid parallelizing them by hand. How can music applications possibly benefit from this brave new world? We answer this question in the following pages.

In many respects music is quite parallel in its structure – as computer scientists would put it, embarrassingly so. Music applications operate on a variety of non-communicating streams variously called voices, channels, lines, or tracks. Most digital audio workstations are organized around this concept and the streams are farmed out to separate threads running on different cores. Examples include the way Ableton Live can use multiple cores and the multi-threading option in Max/MSP `poly~` abstraction. Unfortunately, when using current versions of Max/MSP and Ableton Live together, their multi-threading strategies interfere, and this situation may lead to disruptions in music performance. So even the embarrassingly parallel structure of musical material cannot always be reliably exploited by multiple processors.

Situations like this are difficult to prevent when applications with conflicting runtime requirements execute simultaneously on commodity operating systems such as Mac OS, Windows, or Linux; such systems rarely provide sufficient runtime control or performance isolation. In this paper, we show how to better support these musical applications. Rather than focusing on some new parallel programming language, operating-system concept, or computer architecture, our research effort is driven by the needs of applications themselves.

Popular graphical programming environments, such as Pure Data (Pd) [27] and Max/MSP [10], use directed graphs to represent computations performed on audio streams. One would think that the inherently parallel character of those graphs would afford automatic parallelization. Although there is considerable research on this issue [26, 32], the current widely used versions of Pd and Max/MSP remain essentially sequential. Further, programming real-time musical applications is, for the most part, carried out by the composition of modules (or plug-ins) – a growing number of which are parallel (thanks to component generators like Faust [22]). Thus, assuring that a mix of serial and parallel plug-ins, such as Pd and Max/MSP objects, and VST [33] plug-ins, work efficiently together becomes important. In this paper we discuss how a signal flow graph can be dynamically parallelized and how to meet the needs of parallel plug-ins.

As a unifying theme, we show how a multi-core operating system can be architected to support musical applications. Of particular interest is the fact that musical applications often require simultaneous execution of real-time, interactive, and parallel computations. Tessellation OS [9, 19], our experimental operating system, has two distinctive features: (1) it provides performance isolation and strong partitioning of resources, and (2) it separates global decisions about resource allocation from application-specific scheduling of resources (i.e., two-

Correspondence may be sent to: `juancol@berkeley.edu`.
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. 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.

level scheduling). Central to Tessellation's approach are user-level runtime frameworks, such as Lithe [24], that produce composable, module-specific schedulers. Via Lithe, Tessellation supports a variety of parallel programming models in a uniform and composable way.

In this paper, we also present the design of a real-time parallel musical application on top of Tessellation OS, along with its implementation status and some preliminary experimental results.

2. PARALLELIZING AUDIO DSP GRAPHS

Real-time audio processing applications are often represented by an audio DSP graph (or "audio graph"), a directed tree structure to specify transformations of audio streams. In this section, we discuss techniques for parallelizing such graphs.

Each node in an audio graph represents an *audio module* ("plug-in") that generates one or more channels of output in response to zero or more channels of input. The complexity and computational cost of the audio modules can range from small (e.g., scaling by a constant) to large (e.g., convolution with a long impulse response). Connections between nodes represent the flow of audio streams, and introduce dependency constraints on the order in which modules may be processed. Computations within an audio graph are repeated with the arrival of each new block of input samples, on the order of once every 1ms to 10ms. Presently, most audio applications perform DSP computations within a single high-priority thread in a per audio-stream basis; they are therefore unable to take advantage of multiple cores in processing individual streams.

2.1. Opportunities for Exploiting Parallelism

Using *task-level parallelism* is natural in the execution of an audio graph. Independent modules in the audio graph can be executed in parallel as separate tasks. There are also cases where it is useful for a module to create subtasks that can be executed on different processing elements (e.g., cores), such as in Section 3.1,

Data-level parallelism, on the other hand, can be exploited by dividing the block of input audio samples into segments such that multiple processing elements perform the same transformation on different segments.

The optimal mix of task- and data-level parallelism in the execution of an audio graph is determined by the topology and computational intensity of the elements in the graph, as well as the parallel processing capabilities of the selected hardware platform. Figure 1 shows an example audio graph with both types of parallelism.

2.2. Scheduling the Execution of an Audio DSP Graph

The goal of parallel scheduling is to ensure that the timing requirements of a music application are met while making efficient use of available processing resources. Task-graph parallel scheduling strategies can be static or dynamic.

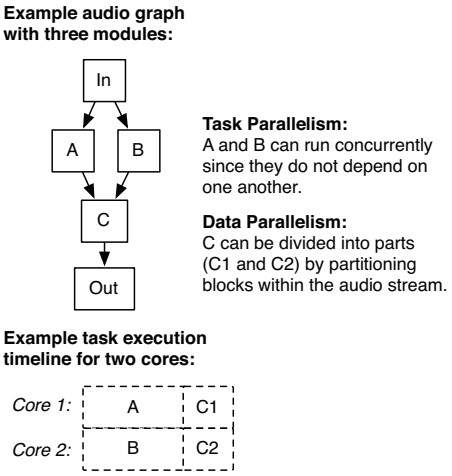

Figure 1. Multi-level parallelism within an audio graph.

A static scheduler creates a complete execution plan before audio streaming begins, while a dynamic scheduler makes online decisions about task ordering and assignment. It is also possible to combine these paradigms, such as choosing a task grouping in advance but scheduling the groups dynamically. Computer music practitioners often prefer dynamic scheduling strategies because they allow for rapid modification of the task graph (i.e., while experimenting or performing).

Dynamic scheduling can be accomplished via *global* or *local* approaches [6]. Global approaches involve a centralized queue of ready tasks that can be executed immediately because they have no remaining dependencies. Processing elements access the central queue to acquire their next task, and upon completion they update the dependency count and activation status of all connected tasks. In contrast, local approaches use a separate queue of ready tasks for each processing element. This eliminates the synchronization penalty incurred by accessing and updating a global queue, but requires load-balancing strategies to distribute tasks evenly among the processing elements. One example of a load-balancing strategy is work stealing [4].

Treating each node in the audio graph as an independent task offers flexibility with parallel scheduling, but in practice suffers relative high overhead. The overhead can be reduced by aggregating multiple nodes into a single executable unit. The goal is to find a sweet spot in the level of aggregation that offers reduced overhead while maintaining enough parallelism to utilize the available processing elements. Any grouping that preserve the dependencies in the original graph is valid. The search space for an optimal grouping can be quite large, but there exist algorithms for finding good approximations [26].

2.3. Task-level Parallelism within Pd Patches

It is useful for computer music developers to utilize multicore processors while programming in familiar languages. We have developed a prototype system that runs compatible Pd patch files and exploits task-level parallelism by

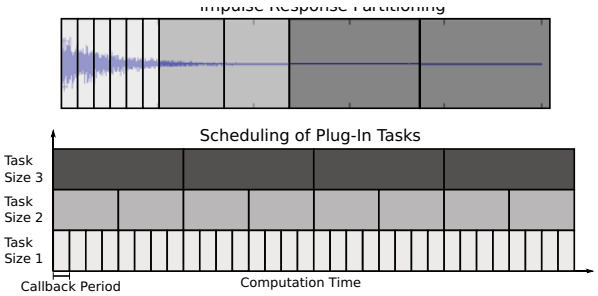

Figure 2. Top - Partitioning scheme for an impulse response using increasing block sizes. Bottom - Scheduling requirements for the tasks for each block size.

assigning the execution of audio modules ("externals") to arbitrary number of processors. It uses either a global or local variety of dynamic scheduler implemented on top of Lithe, a user-level scheduling framework. The system runs on Linux and Tessellation OS (see Section 4), and it is a first step to exploring the concepts presented in the previous sections. We are currently working to support more features of the original Pd runtime, as well as exploring ways to exploit data-parallelism in Pd patches.

3. SUPPORTING PARALLEL AUDIO PLUG-INS

In this section, we discuss support for complex audio plug-ins. Some plug-ins may benefit from parallel execution; in some cases, parallel execution may even be necessary to meet deadlines. Other plug-ins may make use of long-running tasks that are started in one callback (sample period) but do not need to complete until several callbacks later. Supporting parallel execution and long-running tasks within plug-ins complicates scheduling and resource sharing between tasks but is necessary to fully utilize parallel hardware and improve performance.

3.1. Supporting Partitioned Convolution

Partitioned convolution is an important technique for low-latency filtering using long finite impulse response (FIR) filters [13, 3]. It is typically used to simulate, in real-time, the acoustic response of a space (convolution reverb). Partitioned convolution operates by breaking up the impulse response of a long FIR filter into many smaller filters which can be computed in parallel. Chopping up the filter into smaller blocks helps to achieve lower latency; however, smaller blocks are less computationally efficient.

Non-uniform partitioned convolution addresses this problem by allowing the processing block size to increase as we progress through the impulse response. At the beginning of the filter, the smaller blocks allow for low latency; at the end of the filter, larger blocks allow for more efficiency. Further, each of these sub-filters can be computed independently, as separate parallel tasks.

This convolution technique has special scheduling needs, as shown in Figure 2. The non-uniform size of the sub-filters implies that some processing blocks will execute less often than the callback rate, since they must wait for a larger buffer of input samples to be filled. Consequently, we would like to allow tasks to complete in a later callback period than when they are started to avoid forcing large processing blocks to complete within a single callback period.

These special scheduling needs are not exclusive to partitioned convolution. Any plug-in that requires large amounts of computation during each callback would benefit from parallel execution of its tasks, and support for long-running tasks is important to all plug-ins with non-uniform, multi-rate processing needs.

3.2. Supporting Task-level Parallelism

To support task-level parallelism *within* a plug-in, the plug-in developer must insert processing tasks into the scheduler's ready queue(s) and convey task dependencies. In effect the plug-in author provides an *audio sub-graph*, which defines the tasks associated with the plug-in and their dependencies. A potential difference between the main audio graph and the plug-in sub-graph is that the number and arrangement of tasks in the sub-graph may change during execution based on changing parameters or computational requirements. Consequently, it may be important to define an API to handle the plug-in tasks rather than relying on a static graphical arrangement.

3.3. Supporting Long-running Tasks

Adding support for plug-ins with long-running tasks requires adjustments to the basic dependent-task graph scheduler of Section 2.2. A long-running task that is not required to finish during the current callback should not prevent the execution of a task with a deadline in the current callback. Below we present two alternatives for supporting plug-ins of this type.

One alternative is to add task preemption to the task graph scheduler. It allows tasks with sooner deadlines to interrupt long-running tasks. This option is simplest for the plug-in developer; however, preemptive scheduling introduces the overhead of task context-switching.

A second alternative is to divide long-running tasks into time-distributed sub-tasks. The original computation must be carefully divided so that (1) each sub-task can complete during the callback in which it starts, and (2) all sub-tasks are completed by the deadline of the original long-running task. This solution incurs no task context-switch overhead and can yield very high time predictability, but it places extra burden on the programmer and may not even be possible if the task spends much of its time in a call to an external library.

Two authors of this paper (E. Battenberg and R. Avižienis) have created alternative implementations of non-uniform partitioned convolution. Preliminary results suggest that, when running on a single CPU core and using long impulse responses, the preemptive version can achieve between 1.5x-4x more concurrent instances without dropouts than the time-distributed version. For ex-

ample, with an impulse response of 524,288 samples (at 44.1 kHz), the preemptive version reached 82 drop-free instances while the time-distributed version only 22. The test platform was a Mac Pro with two 2.66-GHz 6-core Intel Xeon and 12-GByte RAM running Linux.

A complementary approach is to give the plug-in guaranteed access to a partition of hardware resources (e.g., a set of cores). When access is granted, the plug-in has full control over the partitioned resources. In this case, the plug-in author can use his preferred scheduling scheme, either preemptive or not, to support long-running tasks. Conceptually the plug-in is still part of the audio graph, but it is now an independent server that receives requests from the audio graph. Besides scheduling flexibility, this approach provides performance isolation, which helps the plug-in to deliver consistent performance.

It will become clear in Section 4 that Tessellation OS supports all of the methods described above.

3.4. Supporting External Schedulers

Plug-in developers may try to exploit parallelism by using existing parallel libraries. Usually these libraries are implemented on top of parallel programming frameworks, which use their own schedulers to parallelize computations. For example, Intel's Math Kernel Library (MKL) uses OpenMP [7] to spawn threads. If the main DSP scheduler is unaware of these spawned parallel tasks, the two schedulers will compete for resources. To guarantee cooperative resource sharing and predictable performance, the main DSP scheduler must be able to interface with the external scheduler. Some of these issues are addressed in Section 4.3, which covers a framework for hierarchical cooperative scheduling called Lithe.

4. OVERVIEW OF TESSELLATION OS

Tessellation OS [9, 19] is an experimental multi-core operating system focused on enforcing resource guarantees for client applications. Its development is driven by the needs of real-time audio applications (intended for use in live performances) and other next-generation client applications (e.g., a parallel web browser [14] and a meeting diarist [11]) that can benefit from more computational crunch than a single CPU core can deliver. Tessellation's primary goal is to provide adequate support for a simultaneous mix of real-time, interactive, and high-throughput parallel applications. Other goals include providing scalable performance for parallel client applications and enabling the system to quickly adapt to changes in the application workload and availability of resources.

Tessellation OS is built on two complementary design principles often used in real-time computing [31, 20]: (1) *Space-Time Partitioning* and (2) *Two-Level Scheduling*. Space-Time Partitioning provides performance isolation and partitioning of resources among software components. Tessellation divides the hardware into a set of simultaneously-resident partitions as shown in Figure 3.

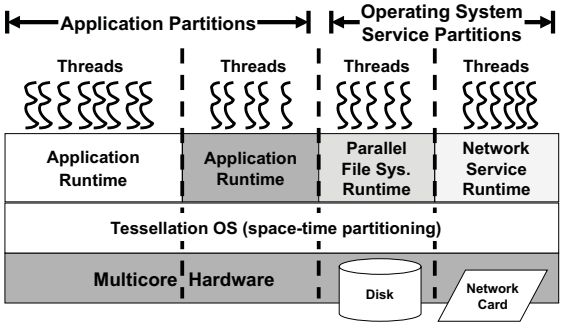

Figure 3. Space-Time Partitioning in Tessellation: a snapshot in time with four spatial partitions.

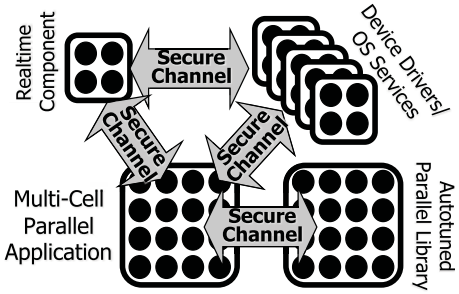

Figure 4. Decomposing an application into a set of communicating components and services running with QoS guarantees within Cells. Tessellation provides Cells that host device drivers and OS services.

These partitions are virtualized and exported to applications and OS services through an abstraction called a *Cell*, which is a container for parallel software components with guaranteed access to resources. An application running within a Cell behaves as it would when executing on a dedicated machine. Two-Level Scheduling, on the other hand, separates global decisions about resource allocation to Cells (first level) from application-specific scheduling of resources within Cells (second level).

4.1. Tessellation Programming Model

Tessellation OS supports a model of computation in which applications are divided into performance-isolated Cells that communicate through efficient and secure channels (see Figure 4). Once resources have been assigned to Cells, user-level schedulers within Cells may utilize the resources as they wish – without interference from other Cells. It is the separation of resource distribution from usage that we believe makes this two-level approach more scalable and better able to meet the demands of parallel client applications than other approaches.

Inter-cell channels provide performance and security isolation between Cells. Channels (once constructed) also provide fast asynchronous communication at user-level, which is an important requirement here.

In addition, partitioned resources assigned to Cells can vary with the needs of the applications and OS (i.e., Cells can "shrink" and "grow"). Tessellation attempts to strike a balance between maximizing resource utilization to achieve performance goals and selectively idling re-

sources to provide quality-of-service (QoS) guarantees. Decision-making logic is packaged into a *Policy Service* that distributes resources to Cells by taking into account system-wide goals, resource constraints, and performance targets combined with current performance measurements. The Policy Service is also responsible for admitting new Cells into the system. For details on Tessellation's Policy Service refer to [9].

4.2. Achieving Performance Isolation

Performance isolation between Cells allows applications to achieve predictable and repeatable behavior, which in turn simplifies performance optimization. Space-Time Partitioning is accomplished through a combination of software and hardware mechanisms. Managed resources include hardware thread contexts, memory pages, and guaranteed fractional services from other Cells, as well as guaranteed portions of shared cache, memory bandwidth, and fractions of the energy budget, when the required hardware mechanisms are available (e.g., [25, 17]).

Cells may be time-multiplexed, as implied by the "time" component of the term Space-Time Partitioning. Hardware thread contexts and other resources are, however, *gang-scheduled* [23] such that Cells are unaware of this multiplexing; in other words, unexpected virtualization of physical resources does not occur. Tessellation implements a variety of time-multiplexing policies for Cells, some of them offering high degrees of time predictability. Examples include the no-multiplexing policy (Cell given dedicated access to cores), time-triggering policy (Cell active during predetermined time windows), event-triggering policy (Cell activated upon event arrivals, but never exceeds its assigned fraction of processor time), and best-effort policy (Cell with no time guarantees). Tessellation incorporates admission control (as part of the Policy Service) and a precedence rule for Cell activation to detect activation conflicts and prevent Cells from compromising the timing behavior of other Cells.

4.3. Lithe: Framework for User-level Scheduling

A Cell-level runtime operates at user-level and manages all resources within the Cell. Central to Tessellation's approach are user-level scheduling frameworks, such as Lithe [24], that enable construction of application-specific schedulers. Lithe enables support for a variety of parallel programming models in a uniform and composable way (see Figure 5).

Via Lithe, applications can effectively utilize one or more parallel libraries without worrying about oversubscription (a situation that occurs when libraries create more threads than there are physical cores in the system). In a standard system, the OS lacks detailed knowledge about how the threads within an application interact; consequently, scheduling decisions often result in decreased application performance. Lithe provides the necessary primitives and interfaces for composing such parallel libraries efficiently.

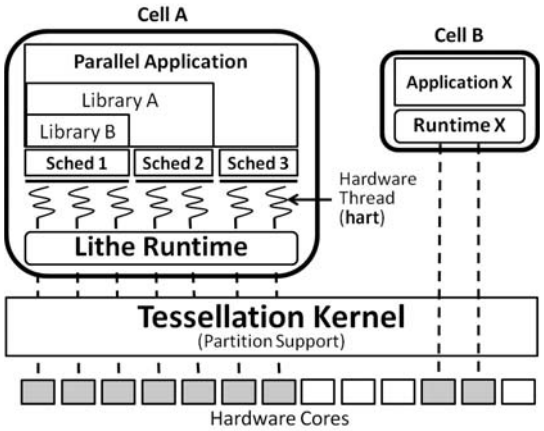

Figure 5. Hierarchical composition of parallel libraries via Lithe within a Tessellation's Cell.

Lithe replaces the virtualized thread abstraction (e.g., `pthreads`) with an unvirtualized hardware thread primitive, or *hart*, to represent a processing element. Applications are given complete control over the management of the harts they have been allocated and are responsible for distributing them among the parallel libraries they invoke. Libraries must be modified to be "Lithe-aware" by replacing calls to traditional threading libraries (e.g., `pthread_create`) with the corresponding Lithe function calls. Lithe-based applications employ hierarchical cooperative (non-preemptive) user-level schedulers to manage the assignment of harts to processing tasks. Applications are therefore built by composing libraries *hierarchically* as shown in Figure 5.

Lithe can be inserted underneath the runtimes of legacy parallel libraries to provide composability without needing to change existing application code. For example, Intel's Threading Building Blocks (TBB) [30], GNU's OpenMP [7] and the FFTW [12] libraries have been ported to Lithe. Lithe can also serve as the basis for building new parallel abstractions and libraries that automatically and efficiently interoperate with one another.

5. A MUSICAL APPLICATION ON TESSELLATION OS

The Tessellation programming model, described in Section 4.1, is a component-based model with composable and predictable performance. Applications can be split into performance-incompatible and mutually distrusting Cells with controlled inter-Cell communication. OS services, hosted in separate Cells, are independent servers that provide QoS guarantees. Computer musicians can take advantage of this structure to make their applications yield high-confidence, predictable performance.

Figure 6 depicts the general structure of a live-performance music application that we are currently developing on Tessellation OS. The application interacts with analog audio inputs and outputs and MIDI controllers via an Ethernet audio I/O device [2]. The device has 10 input and 10 output channels and operates at 44.1kHz. It transmits a packet with a 1280-byte payload (32 samples $\times$ 4 bytes/samples $\times$ 10 input channels) every

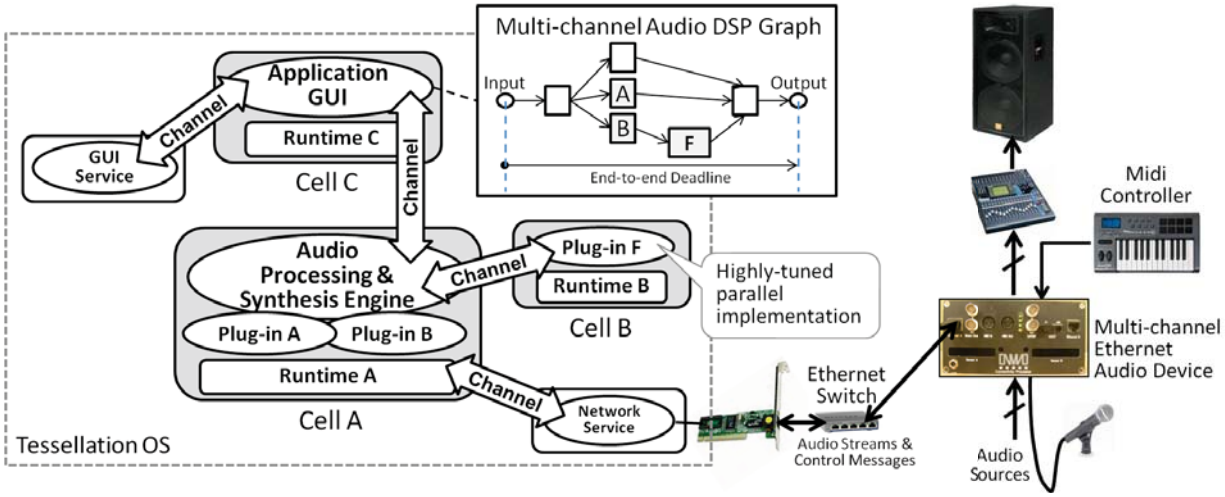

Figure 6. A computer music application deployed on Tessellation OS.

725 μs. The host computer must respond with an equally sized packet of audio output data prior to the reception of the next input data packet to ensure glitch-free operation.

Tessellation's Network Service provides the application a guaranteed communication path to the Ethernet audio device. The Network Service guarantees a minimum communication bandwidth to our application (for both incoming and outgoing messages).

As shown in Figure 6, our proposed music application comprises three Cells:

- *Cell A* hosts the audio processing and synthesis engine (or audio engine) and most of the audio objects and plug-ins defined in the audio DSP graph.
- *Cell B* hosts a highly-tuned parallel plug-in implementing a computationally intensive DSP process.
- *Cell C* contains the GUI application components.

We describe these components in more detail below.

The audio engine in Cell A is central to our application because it controls the execution of the audio graph. It receives input audio data and MIDI control messages from the audio device through the Network Service and returns audio output data. To enable parallel execution of the audio graph, Cell A allocates multiple CPU cores and the user-level runtime within the Cell implements an event-driven deadline-based dynamic scheduler. This scheduler is preemptive and can accommodate long-running tasks without placing extra burden on plug-in programmers (at the expense of some overhead, see Section 3.3). To achieve very low latency, Cell A is not time-multiplexed.

Cell B holds another part of the audio graph, deployed in a separate Cell to avoid interference from the execution of other graph elements. Such highly-optimized parallel plug-ins are likely to require hierarchical composition of parallel libraries. The runtime system in Cell B is thus implemented on top of Lithe. Similar to Cell A, Cell B is assigned multiple processing elements and is not subject to time-multiplexing.

Finally, the GUI components in Cell C allow the user to compose and manipulate audio graphs (e.g., enable and disable specific modules in the graph). Updates to the audio graph are sent to Cell A through a channel. The GUI components interact with the GUI Service, which controls user input and output devices such as display, keyboard, and mouse. This service guarantees high-confidence response times to user events. Cell C uses an event-triggering policy and its runtime implements an earliest-deadline-first scheduler.

5.1. Implementation Status

At this time, we have an early version of the multi-Cell music application, described above, running on Tessellation OS. It includes an audio engine within an independent Cell (i.e., Cell A in Figure 6). The application runs basic audio graphs, defined as Pd patches, and interfaces with the Ethernet audio I/O device, as described above.

Our current Tessellation prototype runs on both Intel x86 platforms and RAMP Gold [34], which is a FPGA-based simulator that models up to 64 in-order 1-GHz SPARC V8 cores. The inter-Cell channels provide asynchronous and lock-free communication at user-level, via a basic version of the Non-Blocking Buffer (NBB) [16, 15] (enabling lock-free communication between a *single* producer and a *single* consumer). Our prototype implements several Cell multiplexing styles, including the no-multiplexing, time-triggering, and best-effort policies; an implementation of the event-triggering policy is underway. Lithe has been ported to Tessellation, providing a framework to construct user-level schedulers. An alternative preemptive scheduling framework is also available. Finally, a basic implementation of the Network Service exists and the GUI Service is in development.

5.2. Experimental Results on Inter-Cell Channels

Fast communication via inter-Cell channels is essential to music applications. A basic requirement is established by the transmission rate of our Ethernet audio I/O device, which is a packet with a 1280-byte payload every 725 μs.

We report the results of two experiments, illustrated in Figure 7. In these experiments we measured the round-

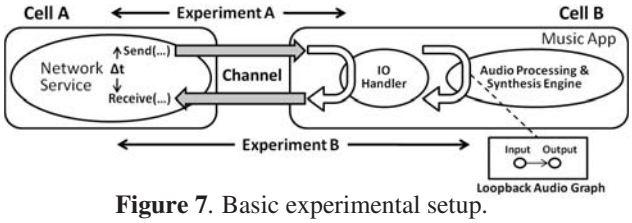

Figure 7. Basic experimental setup.

Message size (bytes)	512	1024	1280	2048	4096	4480
Maximum (μs)	0.85	1.08	1.49	1.62	2.29	3.26
Minimum (μs)	0.67	0.86	0.96	1.27	2.02	2.18
Average (μs)	0.73	0.95	1.08	1.40	2.10	2.42
(CPU cycles)	(1956)	(2542)	(2884)	(3719)	(5599)	(6452)

Table 1. Round-trip times in Experiment A (see Figure 7).

trip times of messages being communicated between two Cells, one hosting the Network Service and the other our application prototype. In each run we collected 100,000 measurements. The system used in our experiments was equipped with a 2.66-GHz Intel Core i7 (quad-core) processor with hyper-threading enabled and 3-GByte RAM.

In Experiment A, we measured the round-trip times of messages with different sizes. The loopback was placed at the door of the application Cell (i.e., no audio graph was involved). Table 1 shows the results. from 100,000 measurements. For messages of sizes above 4 KBytes the maximum observed round-trip time was less than $3.5\mu s$, which means that our channel implementation meets the application's requirement ($\ll 725\mu s$). However, in terms of CPU cycles the results are relatively high because the current channel implementation is very basic and unoptimized. Besides optimizing our current channel implementation, we are investigating message-passing hardware mechanisms that render very fast and efficient, secure inter-Cell channels.

In Experiment B, the audio engine executed a loopback audio graph and each message contained 1,280 Bytes. The average round-trip time was $21.08\mu s$ (56,179 CPU cycles), suggesting that our timing requirement can still be met.

5.3. Experimental Results on Performance Isolation

We conducted a basic experiment to evaluate the quality of performance isolation between Cells in Tessellation. In this experiment we measured the response time of our music application *with* and *without* the presence of another application running in a different Cell. We used the same test platform as in Section 5.2.

As shown in Figure 7, the Network Service and the music application were deployed in Cells A and B, respectively. The music application ran a Pd patch containing eight building blocks regularly found in computer music: a sine-wave generator, a step sequencer, and six classic audio effects. In the audio graph, the eight blocks were parallel to one another, and each of them was applied to the same incoming audio signal and connected to a different physical output. In addition, a third Cell, Cell C (not shown in the figure), hosted a synthetic application. This application continuously ran independent instances of a sorting algorithm on each hardware thread allocated to Cell C.

Application Set	Music App Alone	Music App & Synthetic App
Maximum (μs)	119	121
Minimum (μs)	107	107
Average (μs)	110	110

Table 2. Response times of the music application with and without a synthetic application in another Cell.

The hardware was statically partitioned and the Cells were given dedicated access to specific hardware threads in the system. Cell A and Cell B were assigned one and three hardware threads, respectively. When activated, Cell C was allocated an entire CPU core with its two hardware threads. Paging was not available to the applications.

The application's response time was measured from the instant at which the Network Service sends an input message to Cell B to the instant at which the Network Service receives the output message from Cell B. We ran the music application with and without the synthetic application for 5 minutes and considered the measurements after the first three audio graph executions (warm-up period). As shown in Table 2, there is no significant difference in the response-time values from both scenarios. Hence, with the established hardware partitioning, Tessellation was able to provide sufficient performance isolation to our music application (essentially the performance-isolation level offered by the hardware architecture).

While existing operating systems provide *some* performance isolation via specialized high-priority threads, Tessellation provides general and flexible performance isolation to applications.

6. RELATED WORK

Facilities for exploiting parallelism have been added to computer music environments. The `pd~` object [29] allows one to embed a number of Pd patches, each running on separate threads, into a master patch that handles audio I/O. The Max/MSP `poly~` object [10] generates copies of a patch that are executed by a user-specified number of threads. Faust [22] uses OpenMP or work stealing to automatically parallelize the execution of generated audio components. Moreover, techniques for exploiting fine- and coarse-grain parallelism in Open Sound World (OSW), an environment similar to Pd, have been presented in [8].

Tessellation and other recent many-core operating systems projects, such as Corey [5], fos [35], and Barrelfish [28], share some structural aspects, e.g., distributed OS services. Corey and fos mainly focus on improving OS scalability and, in contrast to Tessellation, do not attempt to provide real-time guarantees. Barrelfish has recently started considering real-time workloads as well, but to our knowledge it has not been used in real-time music applications.

Resource containers, like Linux Containers [18] and OpenVZ [21], offer improved performance isolation

within general-purpose operating systems. We believe, however, Tessellation's features can offer better time predictability and support for parallel workloads. A comparative study of Tessellation and resource containers is part of our future research.

7. FINAL REMARKS

The move to parallelism gives us the opportunity to reinvent the way real-time music and audio software is developed and deployed. Our approach is to let the needs of our applications drive the research effort. Responsive, fault-free, real-time performance is essential to musical applications. Further, composition of programs from components like plug-ins is a productive approach to software development. Thus, our efforts to assure timely behavior in Tessellation OS have yielded a flexible and adaptive approach to structuring parallel musical applications.

As our effort continues, exploitation of performance will play a key role. After all, the move to parallelism is about performance, and most computer musicians appear to have an insatiable appetite for computation.

References

[1] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, E. Lee, N. Morgan, G. Necula, D. Patterson *et al.*, "The parallel computing laboratory at UC Berkeley: A research agenda based on the berkeley view," UCB/EECS-2008, Tech. Rep., 2008.

[2] R. Avizienis, A. Freed, T. Suzuki, and D. Wessel, "Scalable connectivity processor for computer music performance systems," in *Proc. of the Int'l Computer Music Conference*, Berlin, Germany, 2000, pp. 523–526.

[3] E. Battenberg, A. Freed, and D. Wessel, "Advances in the parallelization of music and audio applications," in *Proc. of the Int'l Computer Music Conference*, New York, USA, 2010, pp. 349–352.

[4] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, pp. 720–748, September 1999.

[5] S. Boyd-Wickizer *et al.*, "Corey: an operating system for many cores," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

[6] T. Casavant and J. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering,*, vol. 14, no. 2, pp. 141–154, 2002.

[7] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[8] A. Chaudhary, A. Freed, and D. Wessel, "Exploiting parallelism in real-time music and audio applications," in *Proc. of the 3rd Int'l Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE 99), LNCS 1732*, ser. Lecture Notes in Computer Science, vol. 1732, December 1999, pp. 49–54.

[9] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, Asanović, and J. Kubiatowicz, "Resource management in the Tessellation manycore OS," in *Proc. of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, June 2010.

[10] Cycling'74. [Online]. Available: http://cycling74.com

[11] G. Friedland, J. Chong, and A. Janin, "A parallel meeting diarist," in *Proc. of the 2010 Int'l Workshop on Searching Spontaneous Conversational Speech (SSCS '10)*, 2010, pp. 57–60.

[12] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[13] W. Gardner, "Efficient convolution without input-output delay," *JAES*, vol. 43, no. 3, pp. 127–136, 1995.

[14] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik, "Parallelizing the web browser," in *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, March 2009.

[15] K. H. Kim, J. A. Colmenares, and K.-W. Rim, "Efficient adaptations of the non-blocking buffer for event message communication between real-time threads," in *Proc. of the 10th IEEE Int'l Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, 2007, pp. 29–40.

[16] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, pp. 125–143, March 1977.

[17] J. W. Lee, M. C. Ng, and K. Asanović, "Globally-synchronized frames for guaranteed quality-of-service in on-chip networks," in *Proc. of the 35th ACM/IEEE Int'l Symposium on Computer Architecture (ISCA 2008)*, June 2008, pp. 89–100.

[18] Linux Containers. [Online]. Available: http://lxc.sf.net

[19] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client OS," in *Proc. of the 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, March 2009.

[20] 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.

[21] OpenVZ. [Online]. Available: http://wiki.openvz.org

[22] Y. Orlarey, D. Fober, and S. Letz, "Parallelization of audio applications with Faust," in *Proc. of the 6th Sound and Music Computing Conference*, Porto, PT, 2009, pp. 99–112.

[23] J. Ousterhout, "Scheduling techniques for concurrent systems," in *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 1982, pp. 22–30.

[24] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with Lithe," in *Proc. of the 31st ACM Conference on Programming Language Design and Implementation (PLDI 2010)*, June 2010.

[25] M. Paolieri, 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, pp. 57–68, June 2009.

[26] A. Partzsch and U. Reiter, "Multi core / multi thread processing in object based real time audio rendering: Approaches and solutions for an optimization problem," in *Proc. of the 122th AES Convention, Preprint 7159*, Vienna, Austria, 2007.

[27] Pd. [Online]. Available: http://puredata.info

[28] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," in *Proc. of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar'10)*, June 2010.

[29] M. Puckette, "Multiprocessing for pd," in *Proc. of the 3rd Int'l Pure Data Convention (PDCON09)*, 2009.

[30] J. Reiders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[31] J. Rushby, "Partitioning for avionics architectures: requirements, mechanisms, and assurance," NASA Langley Research Center, NASA Contractor Report CR-1999-209347, June 1999, also issued by the FAA.

[32] R. Sadek, "Automatic parallelism for dataflow graphs," in *Proc. of the 129th AES Convention, Preprint 8259*, San Francisco, USA, 2010.

[33] Steinberg. [Online]. Available: http://www.steinberg.net

[34] Z. Tan, A. Waterman, H. Cook, S. Bird, K. Asanović, and D. Patterson, "A case for FAME: FPGA architecture model execution," in *Proc. of the 37th ACM/IEEE Int'l Symposium on Computer Architecture (ISCA 2010)*, June 2010.

[35] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[36] D. Wessel, R. Dannenberg, Y. Orlarey, M. Puckette, P. V. Roy, and G. Wang, "Panel discussion on reinventing audio and music computation for many-core processors," in *Proc. of the Int'l Computer Music Conference*, 2008.