Virtualized Audio: A Highly Adaptive Interactive High Performance Computing Application

Dong Lu Peter A. Dinda
{donglu,pdinda}@cs.northwestern.edu

Department of Computer Science
Northwestern University

Abstract. The idea behind virtualized audio is to extract sound sources (performers) from their native acoustic spaces and insert them into a virtual space, in which there are one or more listeners. This paper focuses on the insertion step, which is an interactive application that exhibits high computational demands from both aperiodic and periodic real-time tasks, has minimal communication demands, and exposes a significant number of adaptation mechanisms. These properties make it well suited as a driver application for research in interactive high performance computing on shared distributed environments such as clusters and grids. We describe the application, our implementation, and the adaptation mechanisms in detail, and then present an initial performance evaluation of one of its components, and study different server selection strategies for it.

1 Introduction

In traditional audio, the majority of the signal chain from the performers to the listener is executed well before the listener plays a recording. Furthermore, the chain, both before and after the recording, is static. Embedded in the recording are the characteristics of the recording venues, the decisions of the mixing engineers, and the engineers' model of how the recording will be used. Similarly, the reproduction system, which executes the part of the chain extending from the recording to the listener, embodies a static model of the room in which it will be used, and the position and orientation of the listener. These assumptions and their static nature explain why traditional audio is so rarely confused with reality.

It does not have to be this way. The goal of *virtualized audio* is to permit listeners and performers to inject themselves into a shared virtual acoustic space—to let a listener hear what a performer would sound like in his room or in a virtual performance space of his choosing. The listener(s) and performers, recorded or live, are able to move about the shared space at will, the system maintaining the illusion that the performers are in shared performance venue—a guitarist appears to be sitting at your conference table strumming softly.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, and EIA-0130869. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

Achieving this illusion requires solving two problems: interactive source separation, which extracts performers from their performance venue, and interactive auralization, which inserts performers into the listener's virtual performance venue. Final output is to headphones.

While neither source separation nor auralization is a new problem [2, 10, 3, 1, 12, 11, 9, 15], they are new to high performance computing and HPC has much to offer. First, using computational resources beyond those of a single machine, we can compute highly accurate acoustical models based directly on the physics of sound. Second, it is clear that HPC is essential to scaling to arbitrary numbers of performers and listeners. Finally, HPC could make these services "in the network", thus extending virtualized audio down even to wireless handheld devices.

Interactive applications like virtualized audio also offer a challenge to the HPC community in that they require soft real-time service for periodic and aperiodic tasks, which is quite different from the service that traditional scientific applications require. However, virtualized audio also exposes a number of adaptation mechanisms in mapping to underlying resources, as well as the opportunity to trade off between quality and computation/communication across a huge range. This makes answering the challenge possible.

We are currently building a system that implements the forward problem, interactive auralization. In the following, we describe the structure of this system, its implementation, and the adaptation mechanisms built into its design. Then we present preliminary results for scaling and predictive adaptation of the aperiodic component of the system.

2 Application structure

Figure 1 illustrates the structure of our interactive auralization application. There are four components with well defined interfaces between them. The interfaces are designed to maximize the computation to communication ratio for the application.

User-driven immersive audio client

The user-driven immersive audio client is responsible for initiating computation in the other components. This currently implemented as a Windows MFC application. We hope also to develop a wireless PocketPC version that the user would use like a "Walkman" portable headphone stereo device. The client maintains a 3D model of the physical or virtual space being simulated. The model includes the size of the space ("room"), the temperature, and the "walls" in the space. A wall is a triangle with a thickness and material properties. More complex objects can be constructed out of these walls.

The client also maintains a list of sound sources and sinks within the room. Each source is represented by its position in the room and the audio stream associated with it, while each sink is represented by its position. The audio streams are supplied by the streaming audio service.

There is at least one source (a performer) and there are always two sinks (the user's ears). Each source/sink pair represents a computational path for the system. The client displays the space and the source/sink pairs to the user via an OpenGL display. The

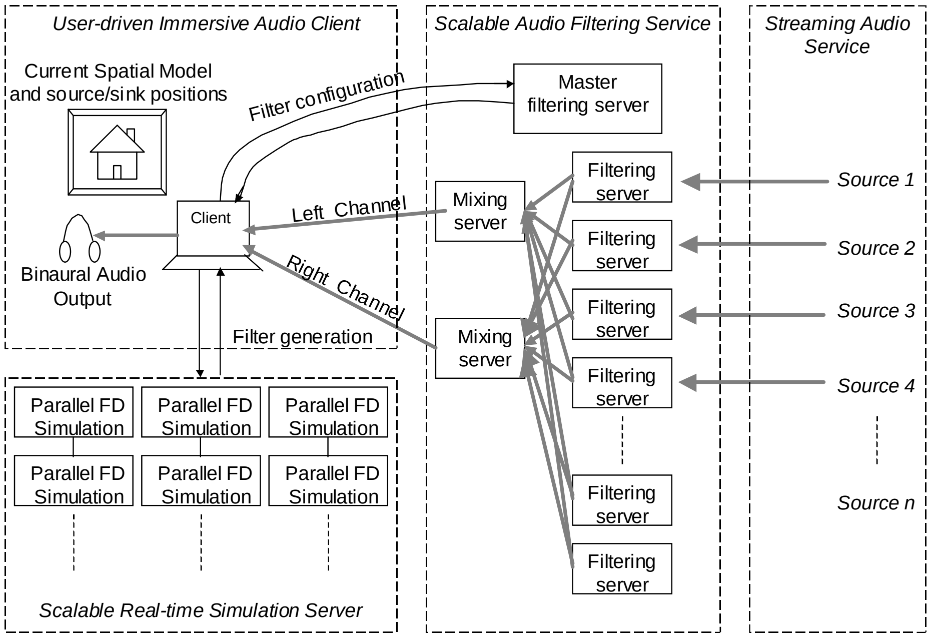


Fig. 1. Structure of interactive auralization.

user can change the room's walls or the positions of sources and sinks at any time. At all times, the client continuously receives and plays, via headphones, two channels of audio from the scalable filtering service, perhaps using a local HRTF filter to model the user's head.

Scalable audio filtering service

For each source/sink pair, the filtering service filters the audio source stream so as to create a new stream that sounds like what the source, when dropped into the room at its designated position, would sound like from the sink's position. For a single source, we would run two filtering servers, one computing a stream for the left ear, and one for the right ear. For each new source, we introduce an additional pair of filters. For a 96 person orchestra, we would run 192 filters simultaneously. A mixing server combines all the filtered streams for a single ear into a single stream for the client. The filtering process runs continuously. It is important to note that a large amount of buffering is not acceptable as the effect of a filter change should be almost immediately apparent.

The filters capture the response of the room from the source to the listener's ears. We assume, reasonably, that this response is due to a linear and time-invariant process. Given this assumption, a filter is the impulse response [14] of the room from the source position to the sink position. Conceptually, if we were to snap our fingers at the source position and record what a microphone heard at the sink position, we would have the impulse response. An impulse response is theoretically of infinite duration, but we limit it in our system. A one second impulse response recorded at the audio CD sampling rate would comprise a vector of 44,100 values. We can use such a Finite Impulse Response (FIR) filter directly, or approximate it using a smaller Infinite Impulse Response (IIR) filter [13].

Streaming audio service

The streaming audio service provides streams of data from the output of the source separation part of virtualized audio—it is the sound of the performer extracted from the original performance venue. At this point, we simply assume that this can be done, and currently use an ordinary digital audio stream from a compact disc. This substitution does not change the auralization process.

Scalable real-time room simulation service

The simulation service is responsible for computing these impulse responses. It does so via a finite difference simulation [16] of the wave equation within the room. The walls are the boundary conditions for the simulation. In effect, the simulation service simulates the finger-snap and microphone scheme described above. The service, which is parallel program written in C++ and PVM, works on a regular 3D grid that has its z-dimension block-distributed across the processors. Each simulation run computes all the filters from a given source simultaneously. The server is stateless: each run is independent.

In steady state, the client simply displays the space and plays the left and right channel streams into the user's headphones. The filtering service applies its filters to the source streams and mixes the results for the client. From the HPC perspective, the challenge is to schedule this collection of periodic soft real-time tasks.

When the room changes, a source or sink position changes, or a new source is introduced, the simulation server is invoked to recompute the effected filters. The common situation is when the user changes his position or orientation, in which case all the filters must be recomputed by the simulation service, and updated in the filtering service. From the HPC perspective, the challenge is to schedule this aperiodic soft real-time task. We will provide some early results on this in Section 4.

Computation and communication requirements

There are two parts to the computational complexity of the system. First, computing the impulse responses from a single source to each sink requires

$$\textit{cost of computing impulse responses of one source} \sim O\left(\frac{xyz(kf)^4 t}{c^3}\right) \qquad (1)$$

stencil operations (about 30 floating point operations each), where xyz is the volume of the room, k is the number of grid points per wavelength, f is the maximum frequency to resolve, c is the speed of sound in the medium, and t is the length of the impulse response. Each computed impulse response is of size $O(kft)$. For air, $k = 4$, $f = 20$

KHz, and $x = 8$, $y = 6$, and $z = 3$ m, this amounts to about 1.38×10^{14} stencil operations for a one second response of size 4×10^4. Clearly this is a challenging problem!

Each filtering server, assuming a simple FIR model, does

$$\textit{cost of filtering} \sim O\left((kf)^2 t\right) \tag{2}$$

multiply-adds per second. This assumes convolution. A filtering server based on frequency domain filtering would trim this to

$$\textit{cost of filtering using FFTs} \sim O\left(kft \log kft\right). \tag{3}$$

In designing the interfaces between the different component, we have tried to minimize the communication in which the client must participate while maintaining the client's centricity. The size of a filtering request is dominated by the filter size,

$$\textit{size of filter} \sim O\left(kft\right). \tag{4}$$

The size of a simulation request is

$$\textit{size of simulation request} \sim O\left(w + 2s\right), \tag{5}$$

where w is the number of walls and s is the number of sources, while the response size is

$$\textit{size of simulation response} \sim O\left(2skft\right). \tag{6}$$

The client receives

$$\textit{audio samples per second from mixing servers} \sim O\left(2kf\right) \tag{7}$$

audio samples per second from the mixing servers. Clearly, the compute/communicate ratio, from the perspective of the client is very high. We expect that this make it possible for the client to be behind a low-speed network connection, or for components to run remotely on a computational grid.

3 Adaptation mechanisms

While the CPU demands of interactive auralization can be daunting, it is important to realize that the application can adapt to available computational and communications resources. The application exposes numerous adaptation mechanisms, and the range over which resource demands can be made to vary is very large. This, combined with the naturally distributed nature of the application and the high compute/communicate ratio makes the application particularly interesting as a distributed adaptive application.

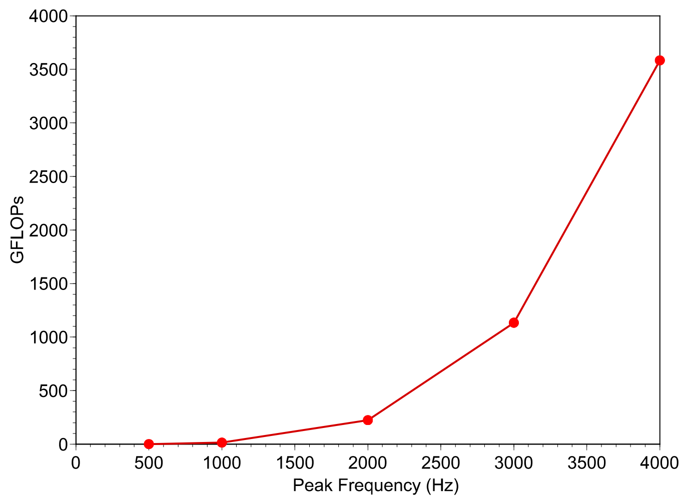


Fig. 2. GFLOPs required to compute a 2 second impulse response on a 8x6x3 space as a function of maximum resolved frequency.

Controlling the peak frequency to be resolved

The most important adaptation mechanism that interactive auralization presents is the maximum frequency that will be resolved, f. As we noted earlier, the amount of computation the simulation service performs to compute a filter depends on f^4. By reducing f slightly, we reduce computation by a large amount. Figure 2 shows the GFLOPs required to compute a single filter as a function of frequency. We limit this and other graphs in the paper to 4KHz (slightly better than telephone quality) due to memory limits in our experimental setup. The memory demands of the simulation service grow as f^3. For the filtering service, the dependence is less dramatic, only f^2. The amount of communication in both services is generally linear in f.

What does this do to quality? Ideally, we would want $f = 20000$, as this is considered the upper range of human hearing by most sources. Most individuals' hearing is considerably more restricted than this, however. FM radio is limited to about 15 KHz, while phones and AM radio are limited to about 3 KHz. Significantly, human perception of pitch is logarithmic in frequency—reducing f by half eliminates one octave, not half of the octaves. Most instruments don't emit a single frequency, however, but rather a set of harmonics that defines their timbre.

Adjusting f by large amounts allows us to map the application to environments with dramatically different computing capabilities. Small adjustments allow us to dynamically adapt to changing resource supply within a particular computing environment. In effect, we trade off computation and accurate reproduction of the pitch and timbre of musical instruments.

Controlling the length of the impulse response

The computation and communication demands of the simulation and filtering services are linear in the impulse response length, t. In principle, the impulse response is of infinite duration. In practice, the length that we choose will be determined by the size and composition of the simulated room. We are trying to capture the first several arrivals of the wave at the sink location. The significance of each subsequent arrival declines quickly. By changing t, we can trade off between computation and capturing the effect that the room has on the sound of an instrument.

Choosing different simulation algorithms

Our simulation server is based on the underlying physics of sound, but other algorithms are possible, and could be plugged into our system. For example, ray-casting approaches [3] or geometry-based approaches [1] could be deployed in environments and under resource-constrained situations where they make sense. For example, they could be used to determine the long-term echoes of very large rooms with less computation than our approach. Another possible approach is precomputation and caching of filters.

Selecting more efficient filters

We can reduce the amount of computation that the filtering service performs, as well as the amount of communication overall by using IIR approximations to the impulse responses that we compute. A linear increase in the number of poles and zeros in the IIR filter makes a more accurate filter at the cost of a linear increase in the amount of communication and computation.

Selecting servers or sites and load balancing

Server or site selection is an important adaptation mechanism. Each time that we invoke the simulation server or add a new filter, we have a choice as to where to run the computation. If the simulation server is asked to compute a very large problem, it can run in parallel, exposing the choice of how many and which machines to use, and raising the possibility of dynamic load balancing of its parallel loop.

4 Experiments

Whenever the user changes the configuration, such as the number or position of sound sources/sinks or the walls of the space, the simulation service is invoked and a new simulation is carried out. As we saw earlier, the simulation is extremely compute intensive, but also high adaptive. Therefore, using parallel or distributed computational resources makes sense. In the following, we evaluate the scalability of the service on a small cluster, and then show interesting initial results for predictive server selection.

Testbed and configuration

Our private testbed consists of eight dual processor 866 MHz Pentium III machines running Red Hat Linux 7.1. Each node has one gigabyte of main memory and a gigabit Ethernet adapter in a 64 bit PCI bus slot. All the nodes are connected to a gigabit Ethernet switch with a 9.6 Gbps backplane. Our evaluation employs a problem where $k = 4$, $x = 8$, $y = 6$, $z = 3$, $t = 2$. For our scaling study, we fix f at 4 KHz. For the server selection experiment, we set f at 500 Hz.

Scalability and the SMP question

Figure 3 shows the speedup and efficiency of the simulation service up to the 16 processors we have available. As the number of processors increases from 1 to 16, we get linear speedup, and the efficiency quickly plateaus. The suggests that the service can scale to the demands that high quality audio will place on it, at least in clusters. The per-processor performance of 20 MFLOP/s is disconcertingly low, however. We do not yet understand why this is the case.

Is it wise to use the second processor on our 2-way SMP machines? In our machines the memory bandwidth is shared by the processors. At first, we attributed what we felt to be the low per-processor performance shown in Figure 4 to the computation being memory bound. If that were the case, then we would expect that using the second processor would not improve performance. To test this, we ran another scalability study, but this time only out to eight processors, adding two processors per step. The processors were either in the same machine ("2-way SMP") or in separate machines ("distributed"). From Figure 4 we can see that although adding separate distributed processors is slightly better than adding a 2-way SMP, there is really not much difference. We conclude that using the second processor on our machines is sensible and that we are probably not memory bound.

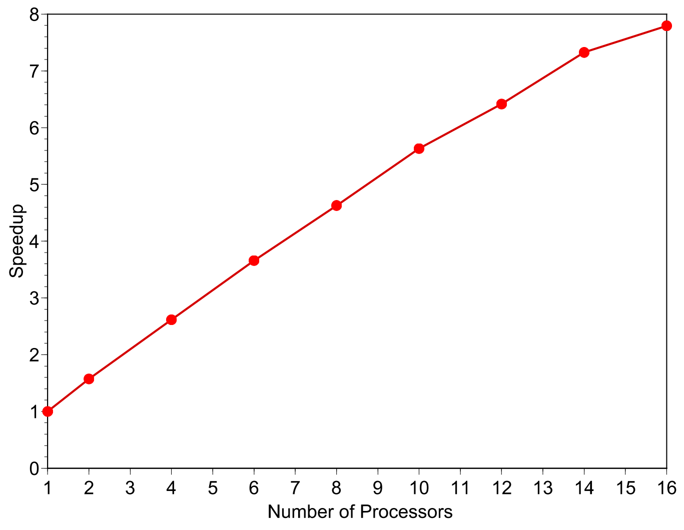
Figure 5 expands on this by plotting MFLOP/s versus peak frequency (f) for a single processor and an SMP pair of processors. While using the second processor does not double performance, it does increase it quite significant. Note that the figure should not be directly compared to the others as we used different compiler optimizations here.

Server selection experiments

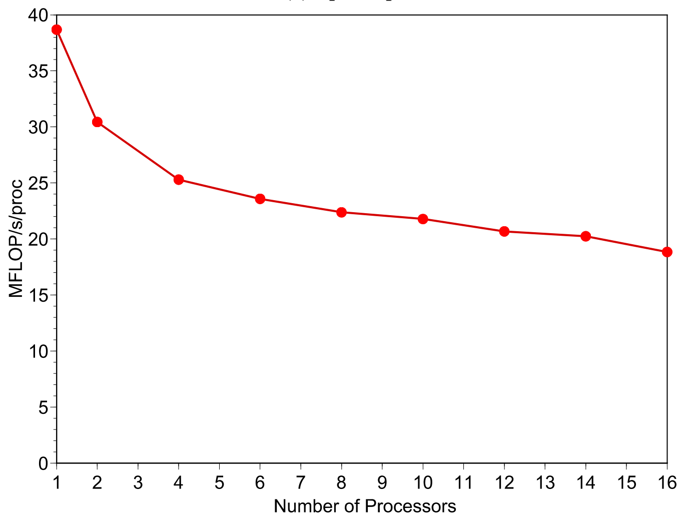
We are particular interested in how the simulation service can adapt to dynamically changing resource availability and have built support for controlling adaptation using our various adaptation tools into the system. In the following, we have chosen f to 500 Hz, so that a request can be serviced on single processor, using about 15 seconds of compute time. The question then is, to which of several available shared servers should we submit this task?

We evaluated four different server selection algorithms implemented using the RPS Toolkit [6]:

- *random*: a random server is chosen,
- *load measurement*: the server with the lowest one minute load average is chosen,



(a) Speedup

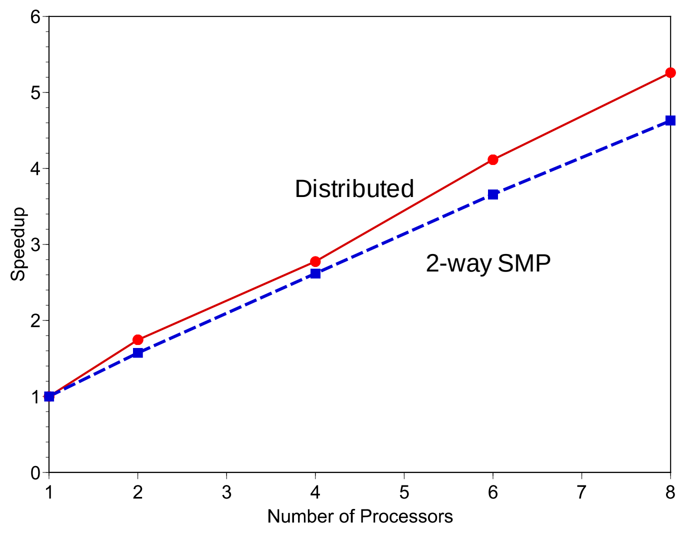


(b) MFLOP/s/proc

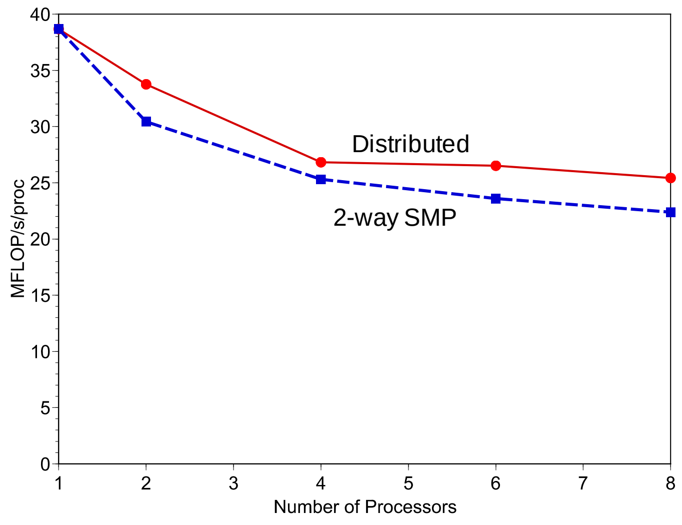
Fig. 3. Speedup and efficiency to 16 processors (8 dual processor nodes).

– *load prediction*: the server with the lowest predicted load [7] over the next 10 seconds is chosen, and

– *RTSA*: our prediction-based real-time scheduling advisor [5] is used to select the server. The RTSA attempts simultaneously to help the client's task meet a deadline and to avoid congestion.



(a) Speedup

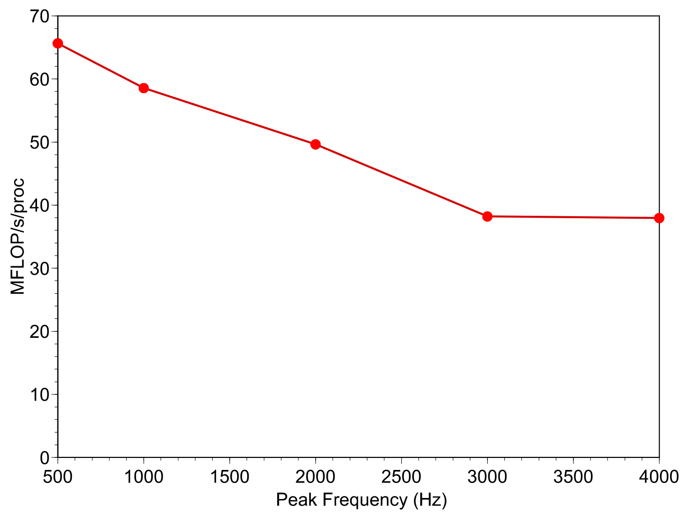


(b) MFLOP/s/proc

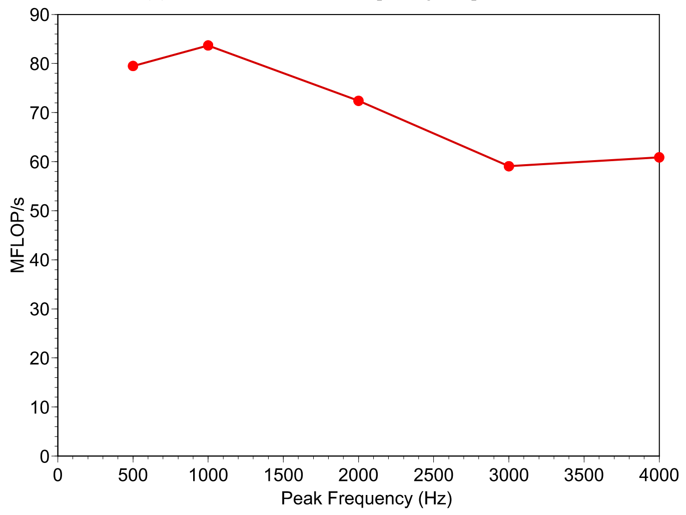
Fig. 4. Performance comparison: 2-way SMP versus distributed memory.

To generate a competitive background workload, we play back [8] load traces [4] collected on a variety of hosts. We disabled the second processor on our servers for these experiments.

We conducted five experiments, the results of which are reported in Figure 6 and discussed below. Each experiment consists of 100 repetitions run at intervals chosen randomly from zero to 30 seconds. Our metrics are the mean and standard deviation of



(a) MFLOP/s versus frequency sequential



(b) MFLOP/s versus frequency for 2-way SMP

Fig. 5. Performance comparison: single processor versus 2-way SMP.

wall clock time and slowdown (wall clock time / task size). The goal is to minimize the mean and standard deviation of both.

Do the algorithms negatively impact performance when there is no competing workload? Experiment zero tries to answer this question by comparing the algorithms on 4 hosts that are all free of load. We can see that all four algorithms perform well - the average slowdown and variation in slowdown is only minimally affected by the more complex algorithms.

		Wall Clock Time		Slowdown	
Description	Algorithm	Average (s)	StdDev (s)	Average	StdDev
EXPERIMENT 0	Random	14.73	0.47	1.00	0.0010
4 hosts with	Load Measurement	14.77	0.46	1.01	0.013
no load	Load Prediction	14.67	0.51	1.01	0.015
	RTSA	14.88	0.49	1.01	0.015
EXPERIMENT 1	Random	24.08	7.14	1.64	0.49
2 hosts with no	Load Measurement	14.94	0.46	1.01	0.011
no load, 2 with	Load Prediction	14.83	0.50	1.01	0.014
high static load	RTSA	15.05	0.21	1.01	0.012
EXPERIMENT 2	Random	21.53	4.32	1.44	0.30
1 host with high	Load Measurement	19.18	2.00	1.26	0.14
dynamic load, 1 host	Load Prediction	17.24	1.45	1.14	0.09
with low dynamic load	RTSA	16.95	1.70	1.09	0.12
EXPERIMENT 3	Random	20.41	6.26	1.38	0.42
4 hosts, each with	Load Measurement	16.68	1.75	1.14	0.097
different low to	Load Prediction	16.54	1.59	1.13	0.090
high dynamic load	RTSA	16.63	1.55	1.14	0.096
EXPERIMENT 4	Random	25.36	3.74	1.72	0.25
4 hosts each with	Load Measurement	23.48	3.99	1.60	0.27
high dynamic load	Load Prediction	23.83	3.38	1.62	0.23
	RTSA	23.99	4.59	1.64	0.29

Fig. 6. Server selection results.

How do the algorithms perform when faced with static, unchanging competing workloads? This is the simplest server selection problem to be faced with. In experiment one we have configured two hosts with no load, and two hosts with constant load (via the Unix command “cat /dev/zero > /dev/null”). We can see that using load information is critical to avoiding congested hosts. Random selection lands us on a heavily loaded machine about 1/2 of the time, leading to a higher average slowdown and standard deviation.

How do the algorithms perform when faced with dynamic load? In experiment two, we have configured two hosts to play back load traces. The first host plays back a trace from a machine with high average load (an interactive machine from the Pittsburgh Supercomputing Center (PSC) Alpha Cluster), while the second host plays back a trace from a machine with a low average load (a batch machine from the same cluster). Because variance of load grows with the average, the first, more heavily loaded, machine is also considerably more dynamic than the second. Experiment two makes the algorithms choose between these machines. Here we see a progression of performance as we add prediction into the mix. Surprisingly, the RTSA is able to bring in the best average performance, even though it is simultaneously trying to avoid congesting either host. The cost of this avoidance is the higher variability in the results.

What happens as we make the set of hosts larger? In experiment three, we are using four servers with dynamic load played back from load traces from the PSC’s alpha cluster. Two of the machines have high average load and dynamicity, while two have low average load and lower dynamicity. Essentially, we have added two more machines like those in experiment two. We can see that performance is roughly the same among the different algorithms, other than random. Essentially, this is an “easier” problem than that of experiment two. As the number of machines grows, it becomes easier to find a machine on which good performance can be achieved, hence the average slowdown and the variation in slowdown both decline.

In experiment four, we have made the problem tougher by having the algorithms choose from four hosts, all of which have a high average load and high dynamicity. Now the algorithms do only slightly better than random selection, and there is no clear winner among them.

5 Conclusion and future work

The principle goal of this paper was to describe an interactive application, virtualized audio, that is both enabled by high performance computing and that presents new challenges to the HPC community. Most exciting to us is that the application exposes numerous adaptation mechanisms that could be controlled by a scheduler or adaptation advisor to provide stable and consistent responsiveness. We presented preliminary results showing the effects of controlling one mechanism, server selection, with different approaches based on resource measurement and prediction.

We intend to continue implementing virtualized audio, albeit slowly given our other current commitments. We also plan to conduct additional research on server selection, adaptation, and dynamic load balancing of the simulation and filtering services.

Acknowledgments

We would like to thank Curtis Barrett, who heroically wrote an initial simplified implementation of virtualized audio in a single quarter, and Sam Benediktson, Aja Nichols, and Jim Trieu, who are helping to implement the current design.

References

1. ALLEN, J. B., AND BERKLEY, D. A. Method for simulating small-room acoustics. *Journal of the Acoustical Society of America 65*, 4 (April 1979), 943–950.
2. BEGAULT, D. R. *3-D Sound For Virtual Reality and Multimedia*. AP Professional, 1994.
3. DHILLON, N. S. Cellular approach for modeling room acoustics: A framework for implementations based on the ray tracing algorithm. Master’s thesis, University of Wisconsin – Madison, August 1994.
4. DINDA, P. A. The statistical properties of host load. *Scientific Programming 7*, 3,4 (1999). A version of this paper is also available as CMU Technical Report CMU-CS-TR-98-175. A much earlier version appears in LCR ’98 and as CMU-CS-TR-98-143.
5. DINDA, P. A. A prediction-based real-time scheduling advisor. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)* (April 2002).
6. DINDA, P. A., AND O’HALLARON, D. R. An extensible toolkit for resource prediction in distributed systems. Tech. Rep. CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

7. DINDA, P. A., AND O'HALLARON, D. R. Host load prediction using linear models. *Cluster Computing 3*, 4 (2000). Earlier version in HPDC 1999.
8. DINDA, P. A., AND O'HALLARON, D. R. Realistic CPU workloads through host load trace playback. In *Proc. of 5th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR2000)* (Rochester, New York, May 2000), vol. 1915 of *Lecture Notes in Computer Science*, Springer-Verlag.
9. FRANK FILIPANITS, J. Design and implementations of an auralization system with a spectrum-based temporal processing optimization. Master's thesis, University of Miami, May 1994.
10. GUPTA, S., VEDULA, S., AND KANADE, T. Incorporation of audio in virtualized reality. Tech. Rep. CMU-RI-TR-00-22, Carnegie Mellon University Robotics Institute, 2000.
11. HUOPANIEMI, J., KARJALAINEN, M., VAELIMAEKI, V., AND HOUTILAINEN, T. Virtual intstruments in virtual rooms — a real-time binaural room simulation environment for physical models of musical instruments. In *Proceedings of the International Computer Music Conference (ICMC '94)* (September 1994).
12. KU, L.-P., AND LEONG, H. W. Optimal partitioning of a rectilinear layout. Tech. Rep. NUS C2/95, National University of Singapore Department of Information Systems and Computer Science, February 1995.
13. OPPENHEIM, A. V., SCHAFER, R. W., AND BUCK, J. R. *Discrete Time Signal Processing*. Prentice Hall, 1999.
14. OPPENHEIM, A. V., WILLSKY, A. S., AND YOUNG, I. T. *Signals and Systems*. Prentice Hall, 1983.
15. SILVERMAN, L. Power, vision, and the death of the radio studio. In *Proceedings of COMPCON Spring '94* (1994), IEEE, pp. 450–453.
16. SMITH, G. D. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Clarendon Press, 1985.