Perceptual Scheduling in Real-time Music and Audio Applications

by

Amar Singh Chaudhary

B.S. (Yale University) 1995
M.S. (University of California at Berkeley) 1998

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Lawrence A. Rowe, Co-Chair
Professor David Wessel, Co-Chair
Professor John Wawrzynek
Professor Ervin Hafter

May 2001

Perceptual Scheduling in Real-time Music and Audio Applications

Abstract

Perceptual Scheduling in Real-time Music and Audio Applications

by

Amar Singh Chaudhary

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Lawrence A. Rowe, Co-Chair

Professor David Wessel, Co-Chair

Academic research of computer music and commercial sound systems is moving from special-purpose hardware towards software implementations on general-purpose computers. The enormous gains in general-purpose processor performance gives musicians and composers richer and more complex control of sound in their performances and compositions. Just as geometric modeling has given graphic designers more control of their scenes and objects, (e.g., independent control of size, position and texture), *sound synthesis* allows musicians more control of musical parameters such as duration, frequency and timbre. Examples of sound-synthesis algorithms include additive synthesis, resonance modeling, frequency-modulation synthesis and physical models. Applications, called *synthesis servers*, allow musicians to dynamically specify models for these algorithms and synthesize sound from them in real time in response to user input. A synthesis server is an expressive, software-only musical instrument.

However, the widespread use of synthesis servers has been frustrated by high computational requirements. This problem is particularly true of the sinusoidal and resonance models described in this dissertation. Typical sinusoidal and resonance models contain hun-

dreds of elements, called *partials*, that together represent an approximation of the original sound. Even though computers are now running above the 1GHz clock rate, it is still not possible to use many large models in polyphonic or multi-channel settings. For example, a typical composition might include eight models with 120 partials each, or 960 partials total. Additionally, current operating systems do not guarantee *quality of service* (QoS) necessary for interactive real-time musical performance, particularly when the system is running at or near full computational capacity. Traditional approaches that pre-compute audio samples or perform optimal scheduling off-line do not lend themselves to musical applications that are built dynamically and must be responsive to variations in live musical performance.

We introduce a novel approach to reducing the computational requirements in real-time music applications, called *perceptual scheduling*, in which QoS guarantees are maintained using voluntary reduction of computation based on measures of perceptual salience. When a potential QoS failure is detected, the perceptual scheduler requests that the synthesis algorithms reduce computational requirements. Each algorithm reduces its computation using specific psychoacoustic metrics that preserve audio quality while reducing computational complexity.

This dissertation describes the perceptual scheduling framework and its application to musical works using additive synthesis and resonance modeling. Reduction strategies are developed based on the results of listening experiments. The reduction strategies and the perceptual scheduling framework are implemented in "Open Sound World," a prototype programming system for synthesis servers. This implementation is then tested on several short musical examples. The computation saved is measured for each example. The quality of the audio output from the servers with and without perceptual scheduling enabled is evaluated by human listeners in a controlled experiment. The results of this experiment have been encouraging. In one example, the average CPU time decreased by about 75%, yet listeners perceived little degradation in audio quality.

The perceptual scheduling framework can be applied to other compute-intensive

algorithms in computer music, such as granular synthesis, pitch detection and sound spatialization. It can also be applied to other perceptually oriented computational tasks, such as real-time graphics and video processing.

Professor Lawrence A. Rowe
Dissertation Committee Co-Chair

Professor David Wessel
Dissertation Committee Co-Chair

"If I were not a physicist, I would probably be a musician. I often think in music. I live my dreams in music. I see my life in terms of music... I get most joy in life out of music."
– Albert Einstein

"I am putting myself to the fullest possible use, which is all, I think, that any conscious entity can ever hope to do."
– HAL9000, *2001: A Space Odyssey*

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to acknowledge my dissertation committee. My co-chairs, Larry Rowe and David Wessel have enthusiastically supported not only the work in this dissertation, but all my research in computers and music as a graduate student. An exchange of e-mail between them five-and-a-half years ago got me started working at the Center for New Music and Audio Technologies (CNMAT), and the rest, as they say, is history. I would also like to thank John Wawrzynek and Ervin Hafter for serving on my committee and offering their advice and support.

This research was supported in large part by the National Science Foundation Graduate Fellowship Program and Gibson Music, Inc. Additional resources were provided by Silicon Graphics, Inc. and the Edmond O'Neill Memorial fund.

Of course, funding alone cannot bring research projects to fruition. In particular, I must acknowledge the contributions of my colleagues at CNMAT, Adrian Freed, Matthew Wright and Richard Andrews, with whom it has been a great pleasure to work over the past five years. Fellow researchers Timothy Madden, Rimas Avizienis and Sami Khoury also contributed to portions of my major research projects at CNMAT, including Open Sound Edit and Open Sound World. I promise I will finally release OSW one of these days!

I would also like to acknowledge several colleagues in the wider academic community. Lippold Haken, from the University of Illinois, and Kelly Fitz, from the University of Washington, helped evaluate the audio examples in my final listening experiment. Their time and advice is greatly appreciated. I would also like to thank Gregory H. Wakefield for his advice, as well as Roger Dannenberg and Miller Puckette for their advice and critiques of early versions of Open Sound World.

Finally, there is the less quantifiable but no less valuable support from family and friends. My parents, Jewel and Banvir, and my brothers, Arun and Ajay, have supported all my academic and musical endeavors for a very long time. Space prevents me from listing all of the people who have contributed to my life in graduate school, but I would like to

acknowledge a few people in particular. David Blackston has provided many years of good friendship and conversation and has always been available to "go out for a beer." Silvia Yee has kept me in touch with my musical side and also well fed with homemade baked goods. And of course there is Leah Fritz, whom I met when I began this dissertation and has made my life immeasurably happier in the time since.

Chapter 1

Introduction

Academic research of computer music and commercial sound systems is moving from special-purpose hardware towards software implementations on general-purpose computers. The enormous gains in general-purpose processor performance gives musicians and composers richer and more complex control of sound in their performances and compositions. Just as geometric modeling has given graphic designers more control of their scenes and objects, such as independent control of size, position and texture, *sound synthesis* allows musicians more control over parameters such as duration, frequency and timbre. Examples of sound-synthesis algorithms include sampling, additive synthesis, frequency-modulation (FM) synthesis, resonance modeling, granular synthesis and physical modeling. Similar advances in pitch-detection and sound-spatialization algorithms provide richer environments for control and realization of synthesized sound. Applications, called *synthesis servers*, allow musicians to dynamically specify models for these algorithms and synthesize sound from them in real time in response to user input. In the hands of experienced users, synthesis servers are expressive, software-only musical instruments.

However, the widespread use of synthesis servers has been frustrated by high computational requirements. This problem is particularly true of the sinusoidal and resonance models described in this dissertation. Typical sinusoidal and resonance models contain

hundreds of elements, called *partials* that together represent an approximation of the original sound. While current computers are fast enough to synthesize individual models of this size in real time, it is still not possible to use many large models in polyphonic or multi-channel settings. Readily available technologies such as Dolby AC-3 audio [123] and open facilities such as the Sound Spatialization Theater at the Center for New Music and Audio Technologies at the University of California at Berkeley (CNMAT) [65] allow musicians the opportunity to create more acoustically live and immersive experiences for their audiences. However, the resources required to synthesize a model on one channel of audio must be duplicated for the other channels. For example, synthesis of resonance models with 120 partials each in an eight-channel audio system requires a total of 960 partials to be synthesized, which saturates a 700MHz Pentium III running the Open Sound World [23] real-time music package under Linux. There is no room left to transform the musician's input gestures into control of the models, or for more sophisticated multi-channel diffusion. An accurate model of an organ pipe may only require twenty to thirty partials, but typical organs have thousands of pipes requiring separate models. Polyphonic synthesis of dozens or even hundreds of models may be required simultaneously. Additionally, in a sound system with fewer audio channels than pipes the synthesis of different pipe models must be dynamically routed to one or more channels to simulate the spatial acoustics of the organ. Even for monophonic instruments, multi-channel audio can enhance the acoustic richness of a model. For example, the diffusion of sound from a saxophone can be simulated using a speaker system such as CNMAT's dodecaphone illustrated in figure 1.1. Because the loudness and timbre of an instrument is different when heard from different angles [18], audio output for each speaker should be synthesized from a different model. A sinusoidal model of a saxophone that contained 80 partials now requires 960 partials to be synthesized. Advanced spatial effects, such as the individual diffusion of ten thousand sinusoidal oscillators to different perceived spatial locations, remain beyond the real-time computational resources of even current 1GHz processors [101]. The shift in professional audio systems

Figure 1.1: The dodecaphone speaker system. A separate speaker is mounted on each face of the decahedron. Audio synthesized from twelve different sound models can be routed to each speaker to enhance the spatial presence and realism of modeled instruments.

---

from CD-quality audio at a 44.1kHz sampling rate with 16-bit resolution to 96kHz sampling rates with 20 to 24-bit resolutions places further strain on computational resources. Not only must a system compute twice as many samples in the same amount of time, but it may also require double-precision floating-point to handle the greater dynamic range and rounding errors on calculations involving sampling rates [5].

Polyphonic synthesis also requires multichannel user input. Little computation is required if the input is from a MIDI keyboard, but if the synthesis is controlled by live musical instruments, pitch detection is required. Polyphonic pitch detection for the six strings of a guitar saturates the computational bandwidth of a 400MHz Apple G3 using the `fiddle` algorithm [94] in the Max/MSP programming environment, leaving no resources for the synthesis being controlled.

Even when there are enough computing resources available to realize an ambitious project, current systems do not guarantee hard real-time performance. Hardware interrupts, network services and other applications compete with real-time music processes, leading to unpredictable increases or decreases in available CPU time. Mathematical exceptions and

memory faults also add unpredictability. If the CPU cannot compute sound samples in real time due to excessive computational requirements or exceptional situations, audio buffers will underflow and gaps or loud "clicks" will be heard in the sound output. While these exceptions may not have the catastrophic effects found in other applications requiring real-time performance guarantees (e.g., rocket launches [3]), they are still unacceptable for live musical performances. Such clicks severely degrade the quality of the experience for performer and musician alike, and increased latency and jitter from unpredictable performance affect the ability of a musician to perform gestures in conjunction with what he or she is hearing.

Traditional methods that pre-compute samples or require off-line analysis to determine optimal scheduling are not applicable because musicians often vary their live performances, not only changing tempi but inserting or removing phrases. Additionally, musicians who use programming environments such as Max/MSP and Open Sound World develop real-time music applications incrementally. Programs are debugged and modified as they are running without intervening compilation and optimization steps, thus preserving run-time state (e.g., pitches and loudness) between incremental modifications. When optimization is required, it is usually done by hand [75].

We introduce a novel approach to reducing the computational requirements in real-time music applications called *perceptual scheduling*. In perceptual scheduling, *quality-of-service* (QoS) guarantees, or guarantees of perceived performance by end users, are maintained in a dynamic system using voluntary reduction of computation based on measures of perceptual salience. When a potential QoS failure is detected, the perceptual scheduler requests running synthesis algorithms to reduce the computational requirements. Each algorithm reduces its computation using specific psychoacoustic metrics that preserve audio quality while reducing computational complexity.

This dissertation describes the perceptual scheduling framework and its application to musical works using additive synthesis and resonance modeling. Reductions in compu-

tation from perceptually scheduled musical examples are measured and the quality of the audio output is evaluated in controlled listening experiments. Additive synthesis and resonance models are used in this research because they can be decomposed into partials that can be easily mapped to both computational bandwidth and audio quality. Additionally, the psychoacoustic contributions of individual partials in these models is well understood. However, the perceptual scheduling framework can be applied to other compute-intensive algorithms in computer music, such as granular synthesis, pitch detection and sound spatialization. It can also be applied to other perceptually oriented computational tasks, such as real-time graphics and video processing [80].

An analogy can be made between perceptual scheduling and other adaptive systems, such as hybrid cars, in which perceived quality (i.e., velocity) is maximized under limited bandwidth (i.e., the limited horsepower of the smaller engine) by dynamic reductions and adaptations to changing performance requirements (i.e., the system dynamically switches the gas engine off at low velocities and switches both the engine and electric motor on during acceleration) [6].

1.1 History and Related Work

This dissertation covers a broad range of fields. The following subsections describe related work in computer-music languages, reactive real-time systems, QoS scheduling, data- and computation-reduction algorithms for audio, and human perception of audio. Related work on sound synthesis is described in detail in chapter 2.

1.1.1 Languages for Musical Sound

One fundamental application in computer music is sound synthesis, or the generation of sound from a program written in a sound synthesis language. Most sound synthesis languages trace their history to the "Music N" languages. In "Music N" languages, primitives called *unit generators* representing oscillators, envelopes, and effects are connected

to form *instruments*, programs that generate sound in response to control parameters. An instrument is applied to a time-ordered sequence of control parameters, called a *score*, to realize a composition. Another important feature of "Music N" languages was the introduction of rates as a language feature. *Sample-rate* computations occur once for each audio sample being generated, while *control-rate* computations occur at a slower rate that allows user input to modify the sound. Examples of "Music N" languages include Music V [77], cmix [66] and Csound [122]. Languages such as Music IV and cmix perform independent sample-rate computations for each sample, which allows the sample rate and control rate to be set independently. On the other hand, *vectorized* languages such as Csound and Music V perform sample-rate operations on vectors of samples and implicitly set the control rate to be the sample rate divided by the vector size. The unit-generator paradigm and rate-based computing have been used in other sound synthesis languages, including the functional language Nyquist [34], the object-oriented language SuperCollider [79], and the MPEG-4 Structured Audio Orchestra Language (SAOL) [109].

With the development of hardware synthesizers supporting MIDI [4], several computer-music languages were developed to process control parameters in real time. Examples of real-time music-event-processing languages include MOXIE [28], MIDI-Lisp [130], Sonnet [60] and Max [92]. Systems such as Max and Sonnet use a visual programming paradigm in which components represented by graphic shapes (e.g., boxes and diamonds) are wired together to form larger structures specifying event flow. A program is represented by a directed graph of these components. Such visual environments can also be used to specify signal flow, as in Ptolemy [70], which is described below. They can also be used by synthesizers with programmable DSPs, such as Kyma [107] in designing and controlling custom sound synthesis. SynthBuilder [89] allowed users to specify instruments visually for mixed hardware and software DSP solutions. However, current desktop and portable computer systems are now fast enough to allow both the sound synthesis and event processing to be computed in real time using only software. This capability has lead to the

development of several real-time software synthesis systems. Max/MSP [136] is a signal processing extension to Max. The languages Pd [93] and jMax [36] are also descended from Max. SuperCollider, "Real-time Csound" [121] and RTcmix [118] allow unit generators to be controlled by external user input in real time. Toolkits such as HTM [43] and STK [29] allow development of custom, high-performance sound synthesis applications in C and C++, respectively, while systems such as Aura [33] define entirely new languages on top of high-performance computation environments.

Open Sound World (OSW) builds on the ideas of these previous languages [23]. It is designed to meet the needs of professional musicians working in live performance or recording situations as well as researchers exploring real-time systems, signal processing, or psychoacoustics. OSW was proposed and developed in response to the experiences of composers and researchers at the Center for New Music and Audio Technologies (CNMAT) at the University of California Berkeley using the Max/MSP environment and the CNMAT Additive Synthesis Tools project [49], which included real-time additive synthesis based on HTM. The OSW project has grown to embrace broader research goals in language and system design.

1.1.2 Reactive Real-time Systems

Research in reactive real-time music synthesis systems draws upon research in specifying and scheduling dataflow networks [71], including ongoing work in the Ptolemy project for specifying different formal models (e.g., synchronous dataflow and process networks). Open Sound World follows the "discrete-event" model in Ptolemy, in which execution flows from one component to another via asynchronous messages. The specification and execution of components in OSW, including the semantics of "activation expressions," is also influenced by work from the reactive programming community, including Reactive C and SugarCubes [115]. In reactive programming, execution is triggered by changes in state, which may then side-effect variables that trigger other executions. Although unusual for a

signal-processing system, this execution model gives OSW programmers a unified view of both continuous audio and discrete events, and allows the use of clocks and audio streams that run at different sample rates. Recent work using audio signals with lower sample rates as control streams suggests that applications can benefit from this unified view [11]. A similar execution model can be found in the computer-music language MOXIE [28], which evaluates procedures in response to events and includes a construct *cause* to evaluate an expression at an arbitrary time in the future.

1.1.3 QoS Scheduling

Previous work in real-time QoS scheduling has largely concentrated on network services [52], but these ideas have been extended to multimedia systems as well [40][17]. In computer-music applications, the service is more homogeneous than the general multimedia and networking cases, but processes and bandwidth requirements can change dynamically and unpredictably, and losses (i.e., glitches in the perceived audio) cannot be tolerated. Rather than schedule unpredictable processes to guarantee QoS, we use a simpler scheduling algorithm and dynamically trade quality for lower computational bandwidth.

The perceptual scheduling framework presented in this dissertation is similar to the adaptive QoS framework described by Li and Nahrstedt [72]. Both systems adapt to QoS variations that arise in heterogeneous computing environments and allow different performance criteria to be specified for different components or applications. In perceptual scheduling, the "system-wide" property is the bounded latency of the audio output and the application-specific properties are the perceptual constraints on how computation can be reduced within each component of the system without compromising audio quality.

1.1.4 Models of Time

Real-time computer-music applications require a model of time that is both flexible and accurate. Flexible timing of events and signals is achieved through the use of a *virtual*

time system, in which time from a clock source is dynamically scaled (i.e., sped up or slowed down), or offset by positive or negative value (i.e., to jump forward or backward). Virtual time is a generalization of a *time map*, or a continuous monotonic function over the duration of a piece of music that maps from one time scale to another time scale [59]. Dannenberg describes the use of time maps for specifying and managing expressive tempo changes in computer accompaniment [32]. Rowe and Smith use a true virtual time system (i.e., the scaling function need not be continuous or monotonic) to synchronize frames of video, audio, still images or other data in media streams [106]. Control of time, such as fast-forward, rewind and pause operations, are expressed as functions of virtual time. For example, fast forward scales the virtual time stream by a factor 2.5, while pause sets the rate to zero so that virtual time does not advance and the player remains on the current frame. The stream can be advanced one frame by offsetting virtual time by one frame period while the rate is zero. Similar virtual time abstractions have been used for expressive control of sound synthesis models in the *softcast* additive synthesizer [48] and Max/MSP programming environment [134]. The virtual time abstraction is built into Open Sound World via components called "time machines" that allow users to construct arbitrarily complex time-manipulation functions. The same virtual time system is used for both events and continuous media.

Virtual time systems require an accurate model of physical time. Accurate time-stamping of discrete musical events is described by Anderson and Kuivila [10]. In their approach, all events must be scheduled to occur at a fixed time in the future. Deadline scheduling is used to execute the events in a temporally accurate sequence, and buffer delay is limited in order to bound latency and minimize jitter in the event stream. This *forward-synchronous* model has been extended to distributed systems by Brandt and Dannenberg [15]. In their system, each node in the system maintains its own clock as a linear function of audio samples it has processed. One node is designated as the master, and global time is set as the number of samples processed by the master times its physical sample

period. The slave nodes use an adjustable virtual sample period to map their individual sample counts to global time. The virtual sample periods are dynamically adjusted using a predictor-controller mechanism to maintain a bounded error between global time on the master and global time on the slaves. The OpenSound Control (OSC) protocol allows optional forward synchronization of events via its timestamping mechanism [131]. However it relies on access to a separate mechanism for clock synchronization, such as Brandt and Dannenberg's system, an AES/EBU master word clock [7], or a network time server using NTP [82] or SNTP [83] protocols.

1.1.5 Data and Computation Reduction in Audio Systems

This research complements the work done in "perceptually lossless" compression [61], particularly the use of masking effects to reduce data bandwidth [138]. The MPEG-1 [111] standard and the MiniDisc [119] use algorithms based on auditory masking to compress time-domain waveform representations. In particular, the algorithm used for the MiniDisc converts a time-domain representation to frequency-domain representations using a modified discrete cosine transform (MDCT) [91] and then quantizes the result based on dynamic sensitivity to frequency ranges and masking characteristics. Masking is also used to hide the quantization artifacts by ensuring that these artifacts are placed temporally and spectrally near sounds that will mask them.

Auditory masking and other measures of perceptual salience have also been used to save computational bandwidth in additive synthesis. Additive synthesis models are composed of *partials* representing sinusoidal components whose sum approximates the waveform representation. These models and algorithms to synthesize waveform representations from them are described in greater detail in chapters 2 and 3, respectively. Marks describes a system that pre-assigns a static weight to each partial and prunes partials by increasing weight if there are not enough computational resources to synthesize the entire model [76]. By contrast, Haken uses a method based on auditory masking to dynamically prune par-

tials from a model during synthesis [53]. This system is constantly making decisions about which partials to prune and which to retain, even in the middle of a note event. We opt for Haken's dynamic approach and build on his work by evaluating efficient methods for pruning partials in real time within a dynamic framework. Model sizes increase or decrease in response to changes in available computational bandwidth. When a potential QoS failure is detected, the pruning methods are switched on to reduce model size and computation. When such a failure becomes unlikely, the pruning is relaxed. In our system, both the pruning decisions and the computation of perceptual salience are dynamic. Dynamic computation of perceptual salience allows the system to better adapt to timbral changes in the model. Such adaptation is necessary if the performer is allowed to modify or switch timbres in real time.

Such a dynamic system is possible because pruning of partials affords *graceful degradation* in quality as a function of the number of partials pruned. Graceful degradation has been a goal of systems for bandwidth management in distributed multimedia as well. A more common strategy for graceful degradation in these applications is multiple description (MD) source coding [95]. In MD source coding, multiple representations of a stream are transmitted, each describing the stream at a reduced quality. Each representation can be received independently to recover the reduced-quality version of the stream. If multiple representations are received, a higher-quality version can be synthesized from them. Thus, the quality of a received stream degrades gracefully if a small number of representations are lost at any given time.

1.1.6 Audio Perception

An important component of this research is measuring the quality of sounds synthesized under reduced computational bandwidth. Wakefield surveys several techniques for measuring the ability of listeners to discriminate between sampled waveform representations from original recordings and synthesized versions [125]. Discrimination techniques measure

the probability of listeners' correctly discriminating between original and synthesized versions of a sound sample. If the probability falls below a specified threshold, the samples are considered "indiscriminable." Other experiments ask listeners to rank sound samples along a unidimensional scale, such as quality. Methods that use pair-wise comparisons among all the sound samples require listeners to only determine which sample is "larger" in the measured dimension. For example, if quality is being measured, listeners decide which of the two samples "sounds better." This method, however, requires $O(N^2)$ listener responses to reconstruct the relative positions of the sound samples along the dimension. An early pilot test of the listening experiments described in this dissertation used this approach, but found that too many judgements were required of listeners and that most of the samples were too close in quality for listeners to discriminate.

An alternative approach allows listeners to assign scores to each sample on a scale with five or more points. Samples can be presented alone or with a reference sound as a comparison. Such a scoring system is used to measure speech quality in communication systems [96]. A similar approach was used in our experiments. Listeners were presented two samples: a reference sample always identified as the "original" and a second sample to be judged. A five-point scale was chosen for compatibility with the mean opinion scores used in evaluation of telecommunications systems [2]. An equivalent scoring system is also described by Moore to measure the effect of data reductions in additive synthesis [84].

1.2 Road Map

This chapter introduced the problems addressed by this dissertation and briefly described related work in other fields. Chapter 2 provides an overview of sound synthesis, focusing on the additive-synthesis and resonance-modeling techniques used in this research. Chapter 2 also presents a brief overview of synthesis server applications. Chapter 3 describes the computation of additive synthesis and resonance modeling and provides performance measurements. Chapter 4 discusses strategies for reducing computation in synthesis algo-

rithms and the results of experiments testing human perception of sound produced using these strategies. Chapter 5 describes the "perceptual scheduling" framework for incorporating reduction strategies into a synthesis server. Chapter 6 presents the results of experiments in which listeners evaluate the quality of musical examples synthesized from a server with perceptual scheduling. Chapter 7 concludes the dissertation and discusses future research directions.

Chapter 2

Sound Synthesis

This chapter introduces sound synthesis and describes several sound synthesis algorithms. It also describes applications called synthesis servers that compute synthesis algorithms in real time. The chapter concludes with a brief description of synthesis algorithms and server technologies that were used for this dissertation.

2.1 Synthesis Overview and Techniques

A sound is commonly modeled a *waveform* signal, that is a function $y(t)$ representing the amplitude of the sound at time t. In digital systems, the continuous time variable t is replaced by a discrete variable n. The resulting digital waveform $y(n/S)$ represents the amplitude *sampled* at time n/S where S is the sampling rate. However, it is common practice to assume the relationship between sample index n, time t, and sampling rate S and refer to the digital waveform as $y(n)$.

Waveform representations of sound are similar to image representations of visual scenes. Images allow efficient transformations on groups of pixels, but very little control over perceptual or geometric features. When such control is required, a vector- or program-based representation (e.g, PostScript) is preferred. These representations are also a form of compression in that they require less space than the images they represent. Sounds can

also be represented by programs, with similar advantages for compression and mutability [108]. The process of generating sound from programs is called *sound synthesis*.

Consider a synthesis function of a pure tone with constant pitch:

$$y(n) = A\cos\left(2\pi f n + \phi\right) \tag{2.1}$$

where the parameters A, f and ϕ represent the peak amplitude, frequency and phase of the sinusoid, respectively. Instead of storing the entire waveform, one need only store the three parameters A, f and ϕ and use them to generate the waveform representation when needed. Controlling the pitch in the functional representation is accomplished by simply scaling the parameter f, whereas changing the pitch of a waveform representation requires complex sample-rate conversion operations to avoid loss of quality (e.g. anti-aliasing to avoid degrading a scaled image) [113].

The parameters of a synthesis function can themselves be time-varying. Thus, we can represent pure tones of variable pitch and amplitude:

$$y(n) = A(n)\cos\left(2\pi f(n) n + \phi(n)\right) \tag{2.2}$$

In this example, the waveform representation has been replaced by three *envelope functions* $A(n)$, $f(n)$ and $\phi(n)$ representing the changing amplitude, frequency and phase, respectively. Envelope functions are usually interpolated between points at a lower and possibly varying sampling rate, called the *frame rate*. The envelope functions act like the control points of curves in graphics, allowing independent control over the different dimensions of the sound. For example, the pitch can be scaled by a factor of two (i.e., raised an octave) without affecting the rate of the sound, as illustrated in equation 2.3 while the rate can be scaled by a factor of two without affecting the pitch, as illustrated in equation 2.4:

$$y(n) = A(n)\cos\left(2\pi(2f(n)) n + \phi(n)\right) \tag{2.3}$$

$$y(n) = A(2n)\cos\left(2\pi f(2n) n + \phi(2n)\right) \tag{2.4}$$

By contrast, time and frequency cannot be scaled independently in a waveform representation. Speeding up the sound raises the pitch by the same factor (i.e., the so-called

"Chipmunk effect"). The frame rate balances mutability of the envelope functions with compression. In other words, a higher frame rate offers more degrees of freedom at the expense of larger representations.

The following subsections describe several synthesis techniques in greater detail. Subsections 2.1.1 and 2.1.2 focus on additive synthesis and resonance modeling, and subsection 2.1.3 describes other commonly used synthesis techniques.

2.1.1 Additive Synthesis

In the previous section, we used a synthesis function that modeled a simple time-varying sinusoid. We can model arbitrarily complex sounds as a sum of sinusoids:

$$y(n) = \sum_{i=1}^{N} A_i(n) \cos\left(2\pi f_i(n) t + \phi_i(n)\right) \tag{2.5}$$

where N is the total number of sinusoids, also called a *partial*. Each partial has its own independent envelope functions $A_i(n)$, $f_i(n)$ and $\phi_i(n)$, which together form a *sinusoidal track*. This sum-of-sinusoids synthesis technique is called *sinusoidal-track modeling* or *additive synthesis*. The envelope functions of all partials share a common frame rate, which is then considered the frame rate of the entire model. Although uniform across partials, the shared frame rate is *non-isochronous* and can vary (i.e., successive frames need not be a uniform distance apart). Throughout this dissertation, we assume that all sinusoidal models have a shared but possibly varying frame rate.

Variations on sinusoidal models include "phase vocoder" models which usually have a fixed number of sinusoids constrained within frequency bands [38], and the more general "McAulay-Quatieri" (MQ) sinusoidal models [78] which are used by most modern computer-based additive synthesizers. In the general models we use, the number of sinusoids can vary between frames, resulting in *birth* and *death* events when a sinusoid begins or ends inside a model.

Similar to the single envelope-controlled sinusoid described in the previous section, sinusoidal models have independent time and frequency parameters. Time can be sped up

Figure 2.1: A sinusoidal track model visualized using Open Sound Edit. Time is horizontal, amplitude is vertical, and frequency increases with depth (i.e., into the page). The yellow lines represent the sinusoidal tracks. The shadows on the ground plane represent the change in frequency as a function of time. The vertical shadows on the moveable window represent the instantaneous frequency and amplitude values of the tracks at the indicated time. Phase is not shown.

---

or slowed down without changing the pitch, and frequency can be scaled without affecting the rate of the model [68]. Thus, unlike a sampled waveform models, a sinusoidal model can synthesize sounds with expressive tempo changes. A user can also "jump" to particular positions in time within a model, or rearrange the order of frames to create new musical material from the model [128]. For example, a model of a sung passage can be converted into a new passage by changing the rate and position of time as the model is being played.

Sinusoidal-track models are most often derived from analyses of real sounds. In phase-vocoder analysis, a sound is passed through a filterbank where each filter analyzes

a small portion of the frequency spectrum. The output of the filterbank is a set of time-varying amplitude and frequency signals that can be used as the scalable envelope functions for additive synthesis [85]. More recent analysis tools, such as those developed by the IRCAM Analysis-Synthesis team [104], produce more general MQ-style models. Loris from the CERL Sound Group further generalizes MQ models by allowing the data points in envelopes to fall in between frame boundaries [42]. Some analysis tools such as SNDAN [13] or Armadillo [74] allow both phase-vocoder and MQ-style models. Visualization and editing tools such as *Open Sound Edit* (OSE), illustrated in figure 2.1, allow analyzed models from a variety of sources to explored visually and modified by composers and sound designers [20]. Sinusoidal models can also be generated algorithmically. For example, classic analog waveforms (e.g., sawtooth, square wave, etc.) and their transformation by analog filters can be efficiently computed from well-known formulae in the frequency domain [19].

Additive synthesis is deeply rooted in both the mathematical modeling of waveforms (i.e., Fourier analysis) and the study of human hearing [124]. Moreover, sinusoidal-track representations are linear in the frequency domain with respect to several important transformations, including independent control of pitch/frequency, amplitude and time scales [97], interpolation between sounds (i.e., morphing) [55], and spectral envelopes (i.e., amplitude as a function of frequency), which makes them easier to control than synthesis techniques such as frequency-modulation (FM) synthesis which map less intuitively into the frequency domain [31]. It should therefore come as no surprise that musicians and musical instrument makers have long been interested in additive synthesis, starting with Thaddeus Cahill's electromechanical *Telharmonium* in the late 1800's [87] and moving to electronic hardware [37] and eventually software [44] implementations. However, the generality and ease of use of additive synthesis comes at the expense of large representations. An accurate model of a sound may require hundreds of tracks and frames. In fact, a highly detailed model may be *larger* than the corresponding waveform representation. Managing births and deaths in variable-size models further complicates synthesis. Consequently, sound syn-

thesis modeling has explored other representations that reduce the space and complexity of a sound specification.

2.1.2 Resonance Modeling

Resonance is the response of an acoustic system to a sound source, called an *excitation*. A *resonance model* represents sounds as an excitation and resonance. Given an excitation waveform $x(n)$, the resonance can be represented as a bank of second-order filters:

$$y_i(n) = a_i x(n) + b_{1i} y_i(n-1) + b_{2i} y_i(n-2) \tag{2.6}$$

$$y(n) = \sum_{i=1}^{N} y_i(n) \tag{2.7}$$

where N is the number of filters in the resonance model, $y_i(n)$ is the response of the ith filter, a_i is the input scaling coefficient for the filter and $b_1 i$ and $b_2 i$ are the feedback coefficients for the previous two samples. The coefficients of each filter can be expressed in terms of more perceptually meaningful parameters: amplitude, frequency and bandwidth.

$$a_i = A_i \tag{2.8}$$

$$r_i = e^{-\pi k_i / S} \tag{2.9}$$

$$b_1 i = 2r \cos 2\pi f / S \tag{2.10}$$

$$b_2 i = -r^2 \tag{2.11}$$

where S is the sample rate of the output waveform and A_i, f_i and k_i are the amplitude, frequency and bandwidth, respectively, of the ith resonant filter.

If the excitation is an impulse, the resonance can be modeled as a special case of additive synthesis in which the frequency and phase of each sinusoid is constant and the amplitude decays exponentially:

$$y(n) = \sum_{i=1}^{N} A_i e^{-\pi k_i n} \cos 2\pi f_i n \tag{2.12}$$

The amplitude A_i determines initial energy of the partial, and the bandwidth k_i determines the rate of decay. A smaller bandwidth means a longer decay, and a bandwidth of zero means the resonance stays at constant amplitude. Since the entire evolution of each decaying sinusoid is defined by just three numbers, resonance models require far less data to specify than general additive synthesis. Any instrument that is struck, plucked, or otherwise driven by a single brief burst of energy per tone can be efficiently modeled using impulse-driven resonances. These instruments include pianos, most percussion instruments, plucked strings, and many of the modern extended techniques for traditional orchestral instruments (e.g., key clicks on wind instruments).

The use of impulse-driven resonance models, as described by Potard et al. [90] follows earlier work using tuned resonant filters to model acoustic resonances of physical systems, including a VLSI-based system by Wawrzynek [126] and the *CHANT* project by Rodet et al. [105]. These systems were used to synthesis both impulsive sounds (e.g., percussion) and non-impulsive acoustic sounds (e.g., flute and human voice) by combining different excitation signals based on nonlinear functions with appropriately tuned banks of resonant filters. However, the resonance-modeling work described in this dissertation focuses only on impulse-driven models that can be modeled with exponentially decaying sinusoids.

Resonance models exhibit the same perceptually meaningful control properties of general additive synthesis models, including independent control of amplitude and frequency. While the decaying-sinusoid resonance models also include independent time scaling, filter-bank models do not support time scaling because they require knowledge of the state of the filters during the previous two samples.

Like sinusoidal-track models, resonance models are often derived from analysis of real sounds. Techniques include fast exponential modeling [88] and matrix pencil methods [67]. The matrix pencil methods are slower but more accurate for short or closely-spaced partials. The compact and perceptually meaningful parameter space of resonance models

also makes them ideal for algorithmic sound-model generation. OSE allows users to construct their own resonance models from algorithmically generated primitives (e.g. beating pairs, harmonic series with $1/f$ amplitudes and decay rates, etc.) [21]. Recent work by Madden and Wessel employs a hybrid technique in which new resonance models are generated by fitting analyzed models to fixed frequency grids [75].

2.1.3 Other Synthesis Techniques

This subsection describes other commonly-used synthesis techniques including granular synthesis, frequency-modulation synthesis, and physical-modeling techniques.

Granular synthesis is an alternative to additive synthesis that is based on a theory developed by Gabor [50]. In granular synthesis, a waveform representation is decomposed into tiny waveform representations, called *grains*, using a Gabor transform or other time-frequency analysis technique [103]. A new waveform is synthesized by selecting a sequence of grains, scaling each grain by an envelope function and combining them in an overlap-add process. If the sequence of grains is exactly the same as the sequence from the analysis, the original waveform representation will be resynthesized. If grains from the analysis sequence are removed without changing the order of the remaining grains, a time-compressed version of the original waveform will be synthesized. If grains are repeated without changing the order (i.e., sequence $G_1, G_2, G_3, \ldots$ becomes $G_1, G_1, \ldots, G_2, G_2, \ldots, G_3, \ldots$), then a time-stretched version is synthesized. Thus, similar to additive synthesis, granular synthesis affords independent control over the duration of synthesized waveform representations. If the length of each grain is proportional to the pitch period (i.e., the inverse of the pitch) of the analyzed waveform, then the pitch of original waveform can be independently scaled as well. This special case of granular synthesis is called *pitch-synchronous overlap-add*, (PSOLA) synthesis [86]. Another variation of granular synthesis uses short-term spectral representations for the grains instead of time-domain waveform representations [102]. The computational cost of granular synthesis is proportional to the number of grains required.

A typical grain size is 20ms, or 882 samples at a 44100Hz sampling rate. A waveform representation lasting one minute would require approximately 6000 grains. Real-time granular synthesis requires less computation than additive synthesis for time-scaling and pitch-scaling applications, a feature that has made it more attractive than additive synthesis for many computer musicians who wish to use these transformations. However, it is possible to improve the efficiency by reducing the number of grains used to synthesize sounds requiring less temporal resolution (e.g., steady-state sounds versus attack transients).

In frequency-modulation (FM) synthesis [25], the phase of a sinusoid, called the *carrier*, is modulated by the output of another sinusoid, called the *modulator*:

$$y(n) = A_c cos(2\pi f_c + \phi_c A_m cos(2\pi f_m + \phi_m)); \tag{2.13}$$

where A_c, f_c and ϕ_c are the amplitude, frequency and phase of the carrier, respectively and A_m, f_m and ϕ_m are the modulator parameters. FM synthesis can be formulated either as a modulation of phase, as defined in equation 2.13, or frequency. The name "frequency-modulation" is used for both formulations. The sinusoidal modulator can be replaced by another frequency-modulated sinusoid (i.e., recursive FM) or a sum of sinusoids. Using only two to six sinusoids, FM synthesizers can produce a wide variety of spectrally rich timbres at a low computational cost. Although FM synthesis algorithms have been widely used in computer music, they do not readily lend themselves to the problems we are addressing because they cannot be easily decomposed into perceptually meaningful components for reduction.

Physical-modeling synthesis techniques attempt to model the physics and playing techniques of acoustic instruments. Digital waveguides [112] use delay lines, amplifiers and filters to simulate traveling waves in air columns, strings and resonating chambers. The Karplus-Strong plucked string algorithm loops a noise burst through a pitch-controlled delay line [64]. "Physically Informed Stochastic Event Modeling" (PhiSEM) is useful for simulating percussive shaker and ratchet instruments [30]. It is included in Cook's STK toolkit [29] and has been ported to Open Sound World. Modal synthesis allows users to syn-

thesize sound directly from the equations describing the movement of vibrating structures as a sum of vibrations, or *modes* [39]. Some physical models incorporate frequency-domain spectral representations or resonant filterbanks (e.g., Wawrzynek's resonant-filter synthesizer, described in the previous subsection), and can potentially benefit from the reduction strategies described in this dissertation.

2.2 Synthesis Servers

Traditional hardware music synthesizers motivated the development of synthesis models and the techniques for controlling them [9][63]. Early digital synthesizers were constructed using custom-designed hardware and most relied on functions such as sampling and frequency modulation that are inexpensive to compute. Because large additive synthesis and resonance models are computationally expensive, they were not widely used in custom-designed hardware. However, modern computers are fast enough to implement a synthesis engine in software. Software synthesizers offer increased flexibility and extensibility compared to hardware synthesizers.

Just as a hardware synthesizer produces sound in response to input controls (e.g., piano/organ keyboard input or another control device), a software synthesizer generates sound output in response to message representing expressive musical input. We call this software synthesizer a *synthesis server*.

As illustrated in figure 2.2, the synthesis server reads sound representations from storage, transforms them in response to user input from a controller, synthesizes the transformed representation and outputs the resulting audio waveform. The storage, control and output for the synthesis server may be part of a single local system, or distributed to several host computers connected by a computer network. Sound representations may be stored on a local disk, network file system or the World Wide Web. The Sound Description Interchange Format (SDIF) is a new standard for storing and transmitting a variety of sound representations that can be used in synthesis servers, including sinusoidal and resonance

Figure 2.2: A synthesis server

---

models [132]. User input may be from widgets in a local user interface, or messages received via MIDI or Open Sound Control (OSC) protocols [131]. Audio output may be written to a local audio device, a disk file or network device.

Examples of dedicated synthesis servers include *softcast* for additive synthesis and resonance modeling [49] and *res* for synthesis of resonance models using filterbanks [45]. Softcast is described in the next subsection. Musicians can also construct their own synthesis servers using a language such as SAOL or Open Sound World, which are described in subsections 2.2.2 and 2.2.3, respectively.

2.2.1 Softcast

Softcast reads sinusoidal tracks from SDIF representations. It can also interpret resonance models as exponentially decaying sinusoids. The models are stored using SDIF. A user-controllable virtual time system, called a *time machine*, controls the temporal rate and position within the model. Modifying the parameters of the virtual time system allows the synthesized sounds to be sped up, slowed down, run backwards, etc. The time machine

Figure 2.3: A diagram of *softcast*, showing the flow of sound representations and control messages. The "timbral prototype" operation interpolates the sinusoidal model for virtual time values supplied by the time machine.

---

and sound model are used to compute instantaneous amplitude, frequency and phase values for each of the tracks, which are then sent through a series of frequency-domain transformations, including transposition, inharmonicity, and dropping partials by frequency range or harmonic number (i.e., drop the third partial or drop all partials outside a 400Hz to 500Hz band). The final output of these transformations is then converted to audio samples by an additive synthesis engine. A schematic of softcast is shown in figure 2.3.

Musicians who need synthesis functions not implemented in a dedicated server like *softcast* can use a sound synthesis language like SAOL or OSW to build a custom synthesis server. Examples of SAOL and OSW synthesis servers implementing the basic functionality

of *softcast* (i.e., synthesizing scalable sinusoidal models from SDIF representations) are described below.

2.2.2 SDIF additive synthesizer in SAOL

In SAOL, users create functional units, called *instruments* and time-stamped lists of control parameters for instruments, called *scores*. Instruments are procedures composed of statements that use standard "C-like" expressions and primitive functions called *opcodes*. Figure 2.4 illustrates a SAOL instrument, `track`, that synthesizes SDIF sinusoidal-track models in an MPEG-4 structured audio stream. This instrument can be used as the basis for a synthesis server. The SDIF representation is embedded in the MPEG-4 stream. As the stream advances, successive frames of the sinusoidal model are read into wavetables. The instrument is alerted to these updates via the `changed` control variable. It then reads the amplitude and frequency values from the wavetables into arrays `amp[]` and `freq[]` (phase values are ignored by this synthesizer). The amplitude and frequency arrays are used to control a bank of oscillators (i.e., `oscil[]`) that synthesizes a waveform representation as a sum of sinusoid functions. The oscillator bank is a primitive data structure in SAOL called an `oparray`. In addition to the built-in functions and data structures, `track` utilizes custom opcodes `getmatrix`, `maketracks` and `get_ind` to convert from the dynamically updated SAOL wavetables to the array representation of the current sinusoidal-model frame. `Getmatrix` and `maketracks` are listen in figure 2.4. `Get_ind` is not shown. This instrument and other SDIF synthesizers implemented in SAOL are described in greater detail by Wright and Scheirer [135].

Although each instantiation of a SAOL instrument represents a single "note event", the note can be arbitrarily long. In this case the note is the entire duration of the sinusoidal model, which is incrementally read and synthesized at the control rate (i.e., "k-rate"), allowing the SDIF data to be modified in real time between successive control rate events.

An efficient real-time implementation of the SAOL instrument can be created using

```
instr track() {
  imports table sdif_table1;
  imports table sdif_table2;
  imports table sdif_table3;
  imports table sdif_table4;
  tablemap tab(sdif_table1, sdif_table2,
               sdif_table3, sdif_table4);
  table mydata(empty,1000);
  table pure(harm,1024,1);
  imports exports ksig changed;
  ksig freq, smfreq, max, f[1024],
       amp[1024], ind[1024];
  asig i, sum;
  oparray oscil[1024];
  // control-rate
  if (changed) {
    getmatrix(tab[changed],1,mydata);
    changed = 0;
  }
  maketracks(f,amp,ind,mydata,max);
  // audio-rate
  i=0; sum = 0; while (i < max) {
    sum = sum + oscil[i](pure,f[i]) * amp[i];
    i = i + 1;
  }
  output(sum);
}

kopcode getmatrix(table t,ivar ind,table mat) {
  // put matrix type #ind from t1 in mat
  ksig i, pos, r, c, ct, x, found, type;
  i = 0; pos = 1; found = 0;
  while (!found) {
    type = tableread(t,pos);
    r = tableread(t,pos+1);
    c = tableread(t,pos+2);
    if (type == ind) { found = 1; } else {
      pos = pos + 3 + r * c;
    }
  }
  tablewrite(mat,0,r); tablewrite(mat,1,c);
  ct = 0; while (ct < r * c) {
    x = tableread(t,ct + pos + 3);
    tablewrite(mat,ct+2,x);
    ct = ct + 1;
  }
}

kopcode maketracks(ksig freq[1024],
  ksig amp[1024],ksig ind[1024],
  table mat, ksig max) {

  ksig i, nr, ix, a, f, ph, k;
  ksig used[1024], oldmax;
  nr = numrows(mat);
  oldmax = max;
  i = 0; while (i < max) {
    used[i] = 0;  i=i+1;
  }
  i = 0; while (i < nr) {
    ix = matread(mat,i,0);
    f = matread(mat,i,1);
    a = matread(mat,i,2);
    ph = matread(mat,i,3);
    k = get_ind(ix,ind,max);
    if (k > max) { max = k; }
    freq[k] = f; amp[k] = a;
    ind[k] = ix; used[k] = 1;
    i = i + 1;
  }

  // manage births and deaths
  i=0; while (i<max && max > oldmax) {
    if (!used[i]) {
       freq[i] = freq[max-1];
       amp[i] = amp[max-1];
       ind[i] = ind[max-1];
       max = max - 1;
    }
    i = i + 1;
  }
}
```

Figure 2.4: SAOL code implementing `track`, an instrument that performs additive synthesis from SDIF-based sinusoidal models. The instrument makes use of auxiliary functions `getmatrix` and `maketracks`. This instrument can be used as the basis for a synthesis server.

the *sfront* SAOL-to-C translator [69].

2.2.3 Additive Synthesis Server in Open Sound World

Open Sound World (OSW), is a "visual dataflow programming language," similar to Max/MSP and the process-network and discrete-event models in Ptolemy. In OSW, primitive components called *transforms* are connected together to form dataflow networks called *patches*, as illustrated in figure 2.5. Patches are themselves transforms, and can be embedded in other patches, allowing abstraction and top-down design. Nested patches create a hierarchical name space in which every parameter of every transform has a unique name. The hierarchical name space affords expressibility not found in pure dataflow programming. For example, several transforms can refer to a single shared resource by its name, even when the transforms and the resource are in separate patches. OSW also includes a strong type system that enforces type compatibility between connected transforms and makes programs more robust. Many types, such as sampled audio signals, are associated with colors in the visual programming environment to aid rapid reading and editing of programs. Examples of color-coded types are described in figure 2.5.

OSW includes a large set of standard transforms for basic event and signal processing, including transforms that manipulate sinusoidal-model and resonance-model representations and synthesize waveform representations from them. Figure 2.5 illustrates an OSW patch containing a *softcast*-like synthesis server. Sinusoidal models are read from an SDIF file into an SDIFBuffer, a shared resource. The SDIFBuffer is given an instance name, timbralproto, that can be used by other transforms to access the resource. Because SDIF files are subdivided into one or more time-ordered lists, called *streams*, the hierarchical name can have a number appended to access a specific stream (more information about streams and the structural details of SDIF files can be obtained elsewhere [132]). The sdif::ToSinusoids transform accesses the first stream in the SDIF buffer using the name timbralproto/1. This transform accepts virtual time values from a TimeMachine transform

Figure 2.5: An additive synthesis server in OSW with inharmonicity and amplitude and frequency scaling. The transform sdif::ToSinusoids performs the same function as the timbral prototype operation in *softcast*, interpolating the sinusoidal model from the SDIF buffer timbralproto using virtual time from the time machine. The remaining inlets on the Inharmonicty and ScaleSinusoids transforms can be used to connect controls from user-interface widgets, MIDI, etc. The blue wires represent sinusoidal models, the red wires represent audio signals, the purple wires represent virtual time and the black wires represent scalar types (e.g., numbers).

(i.e., similar to *softcast*) and outputs a data structure containing instantaneous frequency, amplitude and phase values of the model corresponding to the requested location in time. This data structure is then passed to transforms that perform user-controlled inharmonicity and amplitude- and frequency- scaling operations on the sinusoidal parameters. The result is then sent to the AddSynth transform, which uses additive synthesis to convert the sinusoidal-model representation into a waveform representation, which is then output via

the audio output transform.

The set of available transforms can be easily extended to include more advanced operations. Since the data types passed between transforms are C++ types (i.e., classes or primitive scalars), it is relatively straightforward to add new data types as well. The specification of transforms is described briefly in section 3.2. Details about the real-time scheduling and execution in OSW that are relevant to this research are discussed in that section as well. More information on OSW can be obtained elsewhere [23].

While development systems such as OSW allow users to select only the sound transformations and control structures they wish to use (i.e., as opposed to the large fixed set of functions in *softcast*) and modify them in real time, this versatility comes at a cost. In addition to the overhead of dynamic scheduling, the possibility exists for users to build patches that very quickly exceed the available computational resources. Rather than sacrifice dynamism, as is done in languages such as Aura which statically schedules components on available processors prior to execution [33], we wish to enhance the dynamic scheduling used in OSW to gracefully trade quality for computational bandwidth.

2.3 Sound synthesis used in this research

This dissertation focuses on additive synthesis and resonance modeling because these techniques are easily decomposed into partials. Perceptual quality and computation requirements can both be described as functions of the number of partials in the model. The relationship between reduced computation and audio quality can therefore be more easily quantified than in other synthesis techniques. The potential applications of perceptual scheduling to other synthesis techniques such as granular synthesis and physical modeling are not explored here, but mentioned as opportunities for future research.

OSW is used to implement the synthesis algorithms and reduction strategies developed in this dissertation because it contains several functions and data structures dedicated to efficient additive synthesis and resonance modeling as well as a specialized real-time

profiling tool that measures the execution time of both individual transforms and entire programs. Its open architecture also allows the perceptual scheduling algorithm and its associated computation-reduction procedures to be easily integrated into the system.

Chapter 3

Synthesis Computation

We now turn our attention to the implementation and computation of additive synthesis and resonance models in OSW. Several synthesis algorithms are described, analyzed and measured to determine their computational requirements. The execution and scheduling of OSW transforms is then discussed, including the implementation of reactive real-time quality-of-service (QoS) constraints and potential failures of the current architecture.

3.1 Synthesis Algorithms

This section discusses the computation of additive synthesis and resonance modeling in greater detail, including both computation complexity and measurements of performance on test implementations in OSW.

3.1.1 Additive Synthesis using Oscillators

We present two algorithms for additive synthesis: a traditional oscillator-function approach and an algorithm based on the inverse Fast Fourier Transform (IFFT). The oscillator-based approach uses the definition of additive synthesis presented in equation 2.5:

$$y(t) = \sum_{i=1}^{N} A_i(n) \cos 2\pi f_i(n)t + \phi_i(n) \tag{3.1}$$

where N is the number of partials in the model. A direct implementation requires $\theta(N)$ operations per sample assuming no interframe interpolation of amplitude, frequency or phase. The cosine can be implemented either directly, which is compute intensive, or using a table-lookup function, which is memory intensive. If we ignore the independent phase function, we can reduce the multiplication $f_i(n)t$ to addition by noting that frequency is proportional to the inter-sample change in instantaneous phase:

$$\psi_i(n) = \psi_i(n-1) + 2\pi f_i/S \qquad \text{where } 1 \leq i \leq n \tag{3.2}$$

$$y(n) = \sum_{i=1}^{N} A_i(nS) \cos \psi_i(n) \tag{3.3}$$

where $\psi_i(n)$ is the instantaneous phase of the ith partial at the time of the nth sample. However, this algorithm still requires $\theta(N)$ operations.

OSW includes a transform, AddSynth that implements this algorithm using a table-lookup function. It accepts sinusoidal tracks as input and outputs audio samples. The run-time performance of AddSynth was measured using a patch that contained a sinusoidal model to produce a band-limited sawtooth waveform, as illustrated in figure 3.1. A band-limited sawtooth waveform of pitch f can be approximated as a sum of N sinusoids with frequencies $1f, 2f, 3f, \ldots, Nf$ (where $N < S/2f$) and amplitudes $1, 1/2, 1/3, \ldots, 1/N$. In this test, we generated a waveform of pitch 20Hz because it can be approximated by as many as 1000 partials at a 44100Hz sampling rate.

By running the patch for 60 seconds using successively larger sinusoidal models and measuring the total execution time of each transform using the Profile transform, we were able to plot the performance (measured as average CPU time per audio sample) with respect to the number of sinusoids in the model, as shown in figure 3.2. When running at a 44100Hz sampling rate, the average time to compute each sample must be less than about 20μs (i.e., the sample period minus a small amount of overhead) to remain in real time.

Figure 3.1: A test patch for AddSynth. The AnalogVCO transform creates the sinusoidal model of the band-limited sawtooth wave used for the performance tests. The OneShot transform is used to turn off the patch after 60 seconds have passed.

A 400Mhz Intel Pentium II running Linux was able to synthesize up to 350 partials in real time. As expected, the CPU time of the AddSynth transform, as well as the entire patch, increased linearly with respect to the number of partials. Using these measurements, we estimate a computational cost of 0.053μs per sample per partial. This value, as well as the other per-partial costs stated in this chapter, will be used for predicting computational cost in the perceptual scheduling strategy described in chapter 5.

3.1.2 Transform-domain Additive Synthesis (TDAS)

A more efficient algorithm for additive synthesis, called *transform-domain additive synthesis* (TDAS), involves the use of discrete frequency-localizing transforms such as the inverse Fast Fourier Transform (IFFT). Although other transforms, such as the inverse fast

Figure 3.2: Performance results for the AddSynth test patch. The graph represents performance (measured as μs of CPU time per sample) versus the number of partials in the model. The dotted line above 20 μs represents the threshold between real-time and non-real-time performance on our reference machine when running at a 44100Hz sampling rate.

Hartley transform or discrete wavelet transform can be used as well, only the use of IFFTs will be discussed. Using an IFFT to synthesize "continuous" sinusoidal tracks requires several additional steps, including the use of a "synthesis window" to map the frequencies onto one or more bins (i.e., frequency-domain samples) of a discrete spectrum and an overlap-add process to smooth the discontinuous outputs of successive IFFTs. A TDAS algorithm incorporating these steps is first described in a dissertation by Davis in 1974 [35]. First, the frequency, amplitude and phase values of each sinusoid in the input model must be incorporated into a discrete spectrum to produce a complex-valued array used by the

Figure 3.3: Substitution of AddByIFFT for AddSynth in the test patch.

IFFT. Each sinusoid contributes to the spectrum as follows:

$$X(k) = \sum_{j=1}^{N} A_j(n) e^{i\phi_j} W(f_j/S - k) \tag{3.4}$$

where k is the index of the bin in the spectrum and W is the Fourier transform of the window function being used. Because building a spectrum of size K in this manner requires NK steps, the spectrum is approximated using only a small constant number of large values from W (typically six or eight), resulting in $\theta(N)$ steps. The IFFT of spectrum X is then computed to produce a block of samples $w(l)x(n+l)$, where w is the window function and $0 \leq l < K$. Successive blocks of samples are then added with overlap (i.e., the last $K/2$ samples of one block are added to the first $K/2$ samples of the next block). Because building the spectrum takes $\theta(N)$ steps and computing the IFFT takes $K \log_2 K$ steps, and these steps are performed once for every K samples, the algorithm requires $\theta(N/K + log_2 K)$ steps per sample. For models with hundreds of partials and typical values of K (e.g., 128

Figure 3.4: Performance results for the AddByIFFT test patch. These measurements were taken using an IFFT with 128 bins.

---

bins), this implementation results in a potentially significant reduction, assuming an efficient implementation of the IFFT is available.

An implementation of TDAS was developed by Freed for SGI MIPS platform [44]. On a MIPS R4000 processor, this implementation synthesized approximately 300 partials compared to about 60 partials using oscillator-based synthesis on the same platform [47]. A port of Freed's implementation to Intel processors is available in OSW via the AddByIFFT transform. Like AddSynth, it accepts sinusoidal tracks as input and outputs audio samples. It can be easily substituted into our test patch, as illustrated in figure 3.3.

Using TDAS in the test patch results in a significant performance gain, as illus-

trated in figure 3.4. On the 400Hz Pentium II reference machine, as many as 1000 partials can be computed in real time. Once again, the execution time of both the AddByIFFT transform and the entire patch is linear with respect to the number of partials, and we can estimate a computational cost of 0.016μs per sample per partial. The fixed cost of the 128-bin IFFT is estimated at 1.3μs per sample. Although AddByIFFT requires a greater overhead per sample, it can synthesize more partials per sample than AddSynth, and is therefore preferred for most applications.

A potential disadvantage of TDAS is that a separate instance of the algorithm must be used, and therefore an additional IFFT computed, for each audio output channel. For example, eight instances of AddByIFFT to support eight output channels require approximately 10.6μs per sample to compute the IFFTs. The additional overhead reduces the number of partials computable in real time by approximately 400. However, this still leaves approximately 600 partials computable in real time, which is more than can be computed by a single instance of AddSynth. Consequently, there is no performance benefit to using AddSynth in this situation. However, if we increase the number of channels to sixteen, the time to compute the IFFT's exceed the 20μs available in real time, while sixteen instances of AddSynth can still be used in real time if the total number of partials needed is less than 300.

3.1.3 Resonance Modeling

Resonance modeling can be implemented directly using the filter-bank method described in equations 2.6 and 2.7:

$$y_j(n) = a_i x(n) + b_{1i} y_i(n-1) + b_{2i} y_i(n-2) \tag{3.5}$$

$$y(n) = \sum_{i=1}^{N} y_i(n) \tag{3.6}$$

The N filters and summation require at least $3N$ multiply-add operations per sample. Similar to oscillator-based additive synthesis, this algorithm is linear with respect to the

Figure 3.5: A test patch for Resonators. The SDIF buffer contains the piano model, which is loaded into Resonators using the sdif::ToResonances transform.

---

size of the model. However, we expect to be able to compute more resonance partials than additive synthesis partials using the same computational bandwidth because resonances do not require the additional computation of a transcendental function.

OSW includes a transform Resonators that implements a resonant filter bank. It accepts a resonance model representation (i.e., a set of frequency, amplitude and bandwidth/decay-rate values) and an audio input for the excitation, and outputs the result of the filter-bank calculation as audio samples. A patch that uses an impulse to drive the resonators is illustrated in figure 3.5.

The run-time performance of Resonators was measured using a model of a piano tone, where the number of partials was fixed using the ResLimit transform. In each test, an impulse was sent to Resonators once every second for 60 seconds. During successive executions of this patch, the number of partials was increased (i.e., by increasing the fixed

Figure 3.6: Performance results for the Resonators test patch.

---

limit set by ResLimit). The results are shown in figure 3.6. CPU time increases linearly as the number of resonance partials increases, with a maximum of about 350 partials computable in real time using our reference machine. Because the test patch also continuously executes the Ticker and Impulse transforms to generate a new impulse every second, the difference between the execution time of Resonators and the entire patch is slightly greater than in the additive synthesis examples, where only the synthesis and output transforms were executed continuously. Based on the measured execution times for Resonators, we estimate a computational cost of 0.047μs per sample per partial.

Resonators is a general filter bank that can be applied to any excitation signal. If only the impulse response is needed (i.e., to model struck or plucked instruments), the

resonance model can be converted to an exponentially decaying sinusoidal model using the Res2Sinusoids transform. The output of Res2Sinusoids can then be connected to AddSynth or AddByIFFT to perform additive synthesis, with similar performance results as above. Although the performance of oscillator-based additive synthesis can be optimized for fixed frequencies and exponentially decaying amplitudes [46], such an algorithm has the same computational complexity and does not compete with the performance gained using TDAS except when large numbers of audio output channels are required.

The performance of Resonators was not measured with non-impulse excitation signals. However, the excitation does not affect the number of instructions executed per sample by the resonant filter bank, so no significant difference in performance is expected.

3.2 Execution and Scheduling Issues

We begin the discussion of program execution in OSW by describing transforms in greater detail. A transform is specified as a collection of inlets, outlets, *state variables* and *activation expressions* that a user can view or modify. A state variable is a public variable of a transform that can be queried or modified by other transforms in an OSW patch. Inlets and outlets are special cases of state variables used in connections. An activation expression is a piece of C++ code that is executed when inlets or state variables are modified. It is specified by the variables that will trigger this activation, and the code that should be executed. The specification of Sinewave, a transform that implements a simple sinusoid oscillator is shown in table 3.1.

The activation expression looks like a continuous function of time, or a discrete sample-by-sample computation. However, it is actually computing a vector of samples from a discrete time variable, timeIn. Judicious use of standard C++ in the implementation allows transform writers to be spared the details and complexities of vector-based computation [22]. The state variables NumberOfSamples and SampleRate are *inherited* from a more general class of *time-domain transforms* that manipulate time-domain samples.

Sinewave. Generates a pure tone (i.e., sine wave) signal.

	Name	Type	Default
Inlets	timeIn	Time	
	frequency	float	440.0
Outlets	samplesOut	Samples	
Inherited	SampleRate	float	44100.0
	NumberOfSamples	int	128

Activation Expression activation1, depends on timeIn, no delay

```
samplesOut = sin(TWOPI * frequency * timeIn);
```

Table 3.1: Specification of the Sinewave transform

Execution flows from one transform to another by sending values from an outlet to an inlet and then triggering activation expressions that depend on the inlet. Consider the patch in figure 3.7a. When a user updates the value of the number in the first number-box transform, it passes the number from its outlet to the inlet of the addition transform, triggering an activation expression that adds 2 to the input number and sets the outlet to the sum. The outlet then passes its value to the inlet of the second number box, triggering an activation expression that displays the number on the screen. In figure 3.7b, the Sinewave transform computes a vector of samples using the expression in table 3.1, and assigns the new sample vector to its outlet. The outlet then passes the vector to an inlet of the audio output transform, triggering an activation expression that outputs the samples to audio output device. We leave the explanation of how the activation expression in the Sinewave is itself triggered until section 3.2.2.

3.2.1 The Scheduler

OSW uses a greedy data-driven multiprocessor scheduler for activation expressions. An activation expression is dynamically queued when all dependent variables (i.e., inlets) are changed. Each processor runs a thread that executes queued activations or blocks if there are no activations on the queue:

a) b)

Figure 3.7: Simple OSW patch examples to illustrate execution flow. a) Adding 2 to the value of a number box control. b) Audio output of a pure sine wave.

ACTIVATION-SCHEDULER()
while *true*
do pop an activation expression off the queue and execute it .

In this algorithm, the order of execution within a patch is constrained only by the topology of the graph. For example, in the hypothetical patch shown in 3.8a, the activation expressions of transforms 2 and 3 must be executed after transform 1; transforms 4 and 5 must be executed after 2; and transform 3 after 6. However, there is no constraint on the relative ordering of 4 and 5, 2 and 6, etc., which can be executed in parallel on multiprocessor architectures. Additional constraints can be made explicitly. For example, in figure 3.8 we could require 2 to complete before 3, which we define to imply that 4 and 5 must also complete before 3. Such explicit ordering is required to implement the real-time constraints described in the next section.

When a chain of transforms with no opportunities for parallel execution occurs, as in figure 3.8b, the entire chain is scheduled as a single event. Once transform 1 is activated, transforms 2 and 3 will then be activated in succession without intervention from the scheduler.

In addition to simplicity and the ability to seamlessly utilize multiple processors,

a) b)

Figure 3.8: a) A hypothetical patch with implicit ordering and parallelism. b) A patch with no parallelism that can be scheduled as a unit.

---

this algorithm allows execution to continue while patches are being edited, with activation expressions of new transforms being dynamically scheduled and outstanding activations from deleted transforms being removed from the queue. There is no need for a separate compilation or static-scheduling phase between successive modifications of a patch. Unlike traditional applications programming, in which program creation, optimization and debugging are separate phases, real-time music applications in OSW and similar systems are built incrementally. Initial prototypes are refined by dynamically modifying the patch, with the state of unmodified components maintained during the editing. For example, the waveform output of a synthesis algorithm might be amplitude-scaled to maintain a specific loudness in the acoustic environment. As the user modifies the synthesis algorithm that generates the waveform, maintainance of the amplitude-scaling state is necessary to preserve the perceived loudness. The dynamic scheduling in OSW facilitates this incremental programming style.

3.2.2 Reactive Real-time Constraints

OSW is designed for implementing *reactive real-time* audio and music applications. Reactive real-time involves maintaining output quality while minimizing *latency*, the delay

between input and output of the system, and *jitter*, the change in latency over time [14]. Because of the combination of human sensitivity for jitter and the need for reactive response to gestures, the latency goal for the OSW scheduler is set to 10 ± 1ms [26] [120].

It is the job of the audio output device to implement these QoS constraints, because if the audio output is too late, underflow will occur leading to audible clicks. If it arrives too early, latency increases to unacceptable levels. The audio output device is represented in OSW as a transform that has two state variables that represent these constraints. SampleBufferSize is the *block size*, or number of samples that are sent to the device at once, and TargetLatency is the total number of samples that are allowed to be placed in the output queue awaiting realization by the sound hardware. In order to fulfill real-time requirements, the audio output device has to be able to determine when the signal processing that produces the samples it will output is performed. This is accomplished by controlling *virtual time sources* via a *clock*. Clocks measure "physical" time from hardware devices, such as an audio device or network time source, and have fixed rates and periods. Virtual time is a scalable representation of physical time. Virtual time is handled implicitly by any transform that includes an inlet of type Time (e.g., the Sinewave transform described in the previous section) or explicitly by *time machines*, which scale input from clocks or other time machines.

Although OSW contains several clocks attached to different devices, the default behavior is to use the main audio output device as the main clock. The audio-output device clock implements block-precise timing in which a clock period T is the sample block size divided by the sample rate of the device. The audio output device outputs a block of samples and advances the clock by T via the following activation expression:

```
while(SamplesInQueue() > TargetLatency - SampleBufferSize) {
   DoSomethingUseful();
}
FlushSamples();
clock = clock + SampleBufferSize / SampleRate;
```

Figure 3.9: The audio output device manages real-time latency constraints. The output buffer is currently filled beyond TargetLatency. When it drains below this point, additional samples will be computed until the buffer is refilled. If the buffer drains completely, an audible click is heard.

As illustrated in figure 3.9, updating the value of the clock triggers the virtual time handlers, such as time machines or transforms like Sinewave with time inlets, which then drive several transforms (collectively labeled F in 3.9) that synthesize or process sound to be output during the period. Finally, the audio output activation is scheduled to recur at the end of the period after F has executed. The `DoSomethingUseful` operation is system dependent, and may include deferring to another thread or process to perform other events such as MIDI input. When the output buffer is sufficiently drained (i.e., below TargetLatency), `FlushSamples` outputs the block of samples for this period.

3.2.3 Potential QoS Failures

The dynamic scheduling and real-time-constraint architecture of OSW works well as long as the total execution time for patches remains below the threshold for real-time performance (i.e., 20μs per sample on the reference machine when running at a 44100Hz sampling rate), in which case the system will remain at equilibrium with samples replaced at the same rate they are drained from the buffer. If the execution time is longer, then samples will be drained from the output buffer faster than they will be replaced at a rate

proportional to the sample-computation time. For example, if a new sample takes 40μs to compute, two samples will have already been output from the buffer, so the net number of samples in the output buffer will decrease by one. If the buffer drains completely, there will be a gap in the audio output, perceived as a loud click followed by a brief silence.

Such a QoS failure is unacceptable for music applications, especially during a live performance. Increasing the size of the output buffer for larger, more compute-intensive patches will prevent underflow, but at the expense of increased latency and jitter. Currently, composers and engineers writing patches can manually reduce the computation by removing transforms or decreasing the size of sound models used (e.g., by manually removing components from additive synthesis or resonance models), but such manual reductions may compromise sound quality as well as artists' intentions. They are also hard to do, especially without profiling.

Even in patches where the average per-sample execution time remains below the threshold for real-time performance, the execution time may temporarily increase due to exceptions or interrupts, responses to discrete, non-deterministic real-time events (i.e., from user input) that may activate additional transforms, and dynamic modifications to sound models (i.e., real-time editing or loading of SDIF buffers containing sound models for synthesis).

In the following chapters, algorithms are described that gracefully and dynamically trade audio quality for computational bandwidth. These algorithms can be used to reduce automatically the computation required to maintain QoS guarantees and preserve audio quality and musical intention during large patches, exceptions and dynamic modifications.

Chapter 4

Computation Reduction Strategies

This chapter explores strategies for reducing the computational bandwidth necessary to perform the synthesis algorithms presented in the previous chapter. These strategies focus on reducing the size of the models in order to reduce computational complexity. An acceptable strategy must be incremental, allowing perceptual quality to be gracefully traded for model size and synthesis execution time. Partials must be pruned from the models in a way that minimizes the impact on human perception of the synthesized sound. To this end, partials can be ranked by their *perceptual salience*, or impact on the quality of the sound. They can then be removed in increasing order of salience to reduce size and computation.

A simple method for computing salience is the ranking and removal of partials by increasing amplitude (i.e., the softest partials have the least salience and the loudest partials the most). A measure of salience that better matches properties of the human auditory system is the relative *signal-to-mask ratio* (SMR) [116]. SMR is computed for pairs of partials, called the "masker" and "maskee," by comparing their relative amplitudes and frequencies. Amplitudes are compared on a logarithmic scale (i.e., dB) and frequencies are compares according to a special scale, called the Bark scale, which corresponds to the first 24 critical bands of hearing [137]. Each critical band, or Bark, is approximately 100Hz

Figure 4.1: SMR calculation using masking functions. SMR is calculated as the difference between the height of the softer "maskee" partial and the height of the hat function at its frequency. In the second case, where $SMR < 0$, the softer partial m is masked

---

wide below 500Hz, and the band sizes grow approximately exponentially above 500Hz.* The partial with the higher amplitude is designated the masker. Its masking function is approximated with a "hat" function, whose slopes are 27 dB/Bark towards lower frequencies and 15 dB/Bark towards higher frequencies [51], as illustrated in figure 4.1. The SMR is then computed as the vertical distance between the amplitude of the maskee and the height of the hat function at the maskee's frequency. If the maskee is below the hat function (i.e., negative SMR), it is masked and will not contribute to the sound as heard by human listeners.

For each partial in a frame at time n, we compute its SMR with respect to every

*The published Bark band edges are given in Hertz as 0, 100, 200, 300, 400, 510, 630, 770, 920, 1080, 1270, 1480, 1720, 2000, 2320, 2700, 3150, 3700, 4400, 5300, 6400, 7700, 9500, 12000, 15500.

other partial in the model, and mark it as masked if at least one SMR is negative:

```
FIND-MASKED-PARTIALS(n)
  N ← number of partials in frame at time n
  for i ← 1 to N
    do for j ← i to N
         do if A_j(n) > A_i(n)
              then if SMR(bark(f_j(n)), A_j(n), bark(f_i(n)), A_i(n)) < 0
                     then masked[i] ← true
                     else if SMR(bark(f_i(n)), A_i, bark(f_j), A_j(n)) < 0
                            then masked[j] ← true
```

This algorithm requires $N(N-1)/2$ SMR calculations per frame. This process is optimized by doing no further calculations on a particular partial once a negative SMR has been seen (i.e., if *masked* has already been marked true for the partial), resulting in a data-dependent savings. We then proceed with reduction by first removing the masked partials with negative SMRs and then the unmasked partials with positive SMRs in order of increasing amplitude.

For resonance models, which contain only one "time-independent" frame, a novel optimization is defined for calculating salience based on SMR. Each masker partial M uses a bandwidth-dependent hat function whose left and right slopes are $(6\pi k_M)27$ dB/Bark and $(6\pi k_M)27$ dB/Bark, respectively. Quickly decaying partials with high bandwidth will have steeper hat functions and reduced ability to mask other partials. The factor of 6π is chosen from the definition of bandwidth (i.e., the filter response drops 6dB between the center frequency and a frequency one bandwidth distance away) and mapping of bandwidth to decay rate (i.e., decay rate $= \pi \cdot$ bandwidth).

4.1 Measuring effects of reductions

A group of experienced listeners (i.e., "semi-golden ears") was invited to participate in a series of experiments to measure the effects of our reduction strategies on human listeners. The panel included composers, performing musicians, musical applications re-

Score	Relative Quality	Comparison to original
5	Excellent	Virtually indistinguishable
4	Good	Just perceptibly different
3	Fair	Degraded, but adequate
2	Poor	Strongly degraded
1	Unsatisfactory	Barely comparable

Table 4.1: Scoring system for listening experiments.

searchers and psychoacousticians. Listeners were presented with pairs of sounds with one identified as the original and one identified as a possible reduction. Each listener was asked to rate any perceived difference on a scale from 1 to 5. With these guidelines, panelists were then free to listen to the two sounds in any order as many times as they wanted and to assign scores as they saw fit. The scoring system, illustrated in table 4.1, was chosen to correspond closely to both the 5-point Likert scale used in psychological surveys [73] and the mean opinion scores (MOS) used in the audio/telephony community [2].

It should be noted that panelists were asked not to rate the absolute quality of the sounds (as in mean opinion scores), but rather the comparative quality between the reductions and the original. Early usability testing underscored the need to limit test pairs to comparisons where one is known to be the original.

All listenings were conducted in the recording studio at CNMAT under uniform listening conditions (e.g., speaker levels were calibrated to an 80dB level for a 1000Hz test tone). The experiments were completely automated on a Windows-based PC and double blind, with the investigator only acting as a passive observer and technical support.

Separate experiments were conducted for additive synthesis and resonance modeling, and the results of those experiments are presented in the following two sections. In each case, the results are presented as perceived quality versus model size.

Description	Sound Qualities	Partials	Length
Suling flute	Monophonic, harmonic	150	7.9 seconds
Berimbao	Repeated notes with noisy attacks	200	3.5 seconds
James Brown	polyphonic, multitimbral	250	5.9 seconds

Table 4.2: Sinusoidal-model listening examples.

4.1.1 Sinusoidal models

Three sinusoidal models were chosen, a melody played on a suling flute, a rhythm played on a berimbao (a traditional Brazilian single-stringed instrument), and an excerpt from a 1970 James Brown recording [16], as shown in table 4.2. The analyses of the berimbao and James Brown were performed using the Loris analysis package [42] and the suling was analyzed by the IRCAM Analysis/Synthesis team using their own software [104]. Reductions were made using both the amplitude and standard-masking strategies, yielding 8 reduced models for each strategy at 3/4, 1/2, 3/8, 1/4, 1/8, 1/16, 1/32 and 1/64 of the original size.

Each listening session consisted of six trial groups representing each model and reduction strategy. In each trial group, each of the 8 reduced models plus the original was presented twice for a total of 18 randomly ordered listening examples per trial group, or 108 total examples. The six groups were presented in a random order in one sitting, with strong encouragement to take breaks between groups.

A total of 9 listeners participated in the sinusoidal-model experiment, which was conducted at CNMAT during September and October of 2000.

4.1.2 Resonance models

Three resonance models with different sound qualities were chosen, a marimba, a plucked string bass, and a tam-tam, as listed in table 4.3. Reductions were made using both the amplitude and bandwidth-dependent-masking strategies, yielding 10 reduced models for

Description	Sound Qualities	Partials	Length
Marimba	Harmonic, noisy attack, short decay	48	1 second
Plucked string bass	Strongly harmonic, low frequencies, long decay	59	4 seconds
Tam-tam	Inharmonic, long decay, "noisy" high frequencies	183	8 seconds

Table 4.3: Resonance-model listening examples.

each strategy at 3/4, 5/8, 1/2, 3/8, 1/4, 3/16, 1/8, 3/32, 1/16 and 1/32 of the original size. All the models were treated as exponentially decaying sinusoidal models (i.e., the resonance models were impulse driven).

Each listening session consisted of three trial groups representing one of the two strategies (chosen randomly) for each of the models. In each trial group, each reduced model was presented along with the original three times, plus three instances of the original paired with itself, for a total of 33 randomly ordered listening examples per group, or 99 total.

For the resonance-model tests, which were conducted earlier than the sinusoidal-model tests, panelists were allowed to assign a score between 1 and 8. These scores were then converted to the 5-point scale described above for analysis. A total of 12 panelists participated in this experiment, which was conducted at CNMAT during June and July of 2000.

4.2 Results

The results for the sinusoidal- and resonance-model experiments are presented separately.

4.2.1 Sinusoidal Models

The results for the sinusoidal model experiments, expressed as average user rating versus number of partials, are shown in figures 4.2, 4.3, and 4.4. We consider a mean score of 4 or above to be high quality. All three models are rated consistently high with a steep drop-off in perceptual quality for very reduced models. In each case, the steep decline in quality can be attributed in part to the increase in the number of birth and death events in the model, which occur because a partial may have a different salience in each frame. While the birth and death of a partial with low salience has little effect on the sound output, the effect for more salient partials is dramatic, with loud artifacts that make the sounds seem even more degraded than if the partial was left out entirely.

Although the results exhibit a similar shape, the onset and rate of the decline in quality does differ among the three examples. The suling model showed the steepest drop, which occurs when the model is reduced below 9 partials (i.e., 1/16 of the original size). The suling sound is strongly harmonic with most of the energy concentrated in five partials, allowing very small reductions to retain most of the sound quality. More surprising is how well the "breathy" quality of the sound is maintained at higher reductions, implying that the noise energy is concentrated around a few frequencies. By contrast, the energy in the spectrally richer berimbao model is spread over a larger number of partials, so more of the sound is removed from the model at larger reductions. Moreover, the attack of each note is characterized by strong transients. Removing the partials responsible for these transients noticeably reduces the quality of the sound. The James Brown example includes not only multiple notes but multiple instruments (i.e., voice, drums, electric guitar and bass, and horns) with complex overlapping spectra. While the voice and horns remained recognizable at higher reductions, the rhythm section instruments, and the percussion in particular, lost most of their spectral richness. There was also more change in the spectrum between successive frames because of the timbral variety and note changes. This caused the reduction algorithm to choose different partials in successive frames and produce more

Figure 4.2: Results for reductions on the suling model. In each graph, the x axis is the number of partials in each reduction and the y axis is the average user rating for each reduction. Error bars represent standard deviation, and are removed from the comparison graph for clarity.

Figure 4.3: Results for reductions on the berimbao model.

Figure 4.4: Results for reductions on the James Brown excerpt.

birth and death artifacts.

We can measure the "amount of sound removed" quantitatively as the ratio between the energies, or sum of amplitudes, in the reduced frame and the sum of amplitudes in the original frame:

$$\frac{\sum_{i=1}^{N_R} A_i(n)}{\sum_{i=1}^{N} A_i(n)} \tag{4.1}$$

where N and N_R are the number of partials in the original and reduced frames, respectively, at time n. Using this formula, we see that when the berimbao begins to show strong degradation at 50 partials, on average 68% of the energy remains in each frame, while approximately 50% of the energy remains in the James Brown example at 60 partials (the point at which the quality begins to drop dramatically). We therefore conservatively estimate that such models can be reduced to approximately two-thirds of their original energy independent of the number of partials removed. The suling model retained 98% of its original energy when reduced to just 5 partials, suggesting that this reduction would be rated higher by listeners were it not for the artifacts introduced by frequent births and deaths.

Another significant result is the strong similarity in perception of the amplitude and masking strategies. This result may seem at odds with the results of masking-based strategies used in perceptual encoders for compression. However, it is important to note that perceptual encoders remove masked frequencies from evenly-spaced samples in a full spectrum (i.e., the result of a Fourier Transform), while the frequencies in sinusoidal models are already distilled from full spectra via peak picking [110]. On average only 25% of the partials in each model were masked

4.2.2 Resonance Models

The results of the resonance model experiments are shown in figures 4.5, 4.6, and 4.7. The marimba resonance model exhibits a steep drop in quality below 7 partials (i.e., less than 1/8 of the original model size), which is similar to results for the sinusoidal models. However, the bass and tam-tam models show a more gradual decline in quality as partials

are removed. Both the bass and tam-tam models received scores below 4 when more than half the partials were removed.

The marimba model is quite similar to the suling model in that relatively few partials contribute most of the energy in the resonant, harmonic portion of the sound. Indeed, only 7 partials contributed 93% of the initial energy (i.e., the sum of initial amplitudes). Most remaining partials represent the extremely brief attack at the beginning of the sound. Because no one partial has a strong spectral contribution to the attack, most can be removed without a strong effect on the sound. By contrast, the bass and tam-tam models are characterized by spectrally richer tones that decay over a longer period of time. In particular, most of the partials in the bass model contribute to a single harmonic series with additional partials as "beating pairs" for the harmonics below 900Hz. Removing the harmonic or beating frequencies greatly reduces the sound quality. In the tam-tam model, a large number of high-frequency partials contribute to the brightness and roughness of the sound. However, they were more likely to be removed from the reductions because each of these partials has a relatively low amplitude and each partial was likely to be masked by another nearby partial. The removal of these partials gave the reductions below 50% a "duller" timbre.

An interesting phenomenon observed by many listeners was a perceived change in pitch between the original tam-tam and the significantly reduced versions, which sounded almost a quarter-step lower. Pitch in complex inharmonic tones is often determined by attempting to fit one or more harmonic series to the partials, resulting in weak or ambiguous pitches [117]. Removing a partial may change the relative strength of the harmonic series fit to the sound, causing the pitch to become more ambiguous or even shift.

As with the sinusoidal models, the results of amplitude and masking strategies are remarkably similar. While the reduced models of the marimba were nearly identical in both strategies, the equivalent-sized amplitude and masking of the bass and tam-tam only had 80% and 50% of partials in common. Although nearly 50% of the partials in the

Figure 4.5: Results for reductions on the marimba model.

Figure 4.6: Results for reductions on the plucked string-bass model.

Figure 4.7: Results for reductions on the tam-tam model.

tam-tam were masked using the bandwidth-enhanced method, most of these partials appear to have been masked by other low-amplitude partials, minimizing the difference in results between the two strategies. A pilot run of the experiment used conventional masking based on initial amplitudes (i.e., no decay-rate information), but the results showed even faster degradation, particularly for the bass model because short partials were masking longer partials with lower initial amplitudes.

4.3 Developing a Reduction Algorithm

Evaluating the reduction strategies requires weighing the perceived quality of reduced models against the computational cost that will be added to programs to perform the reductions. Amplitude-based reduction requires that all partials be sorted by descending amplitude, which is an $\theta(N\log N)$ operation where N is the number of partials. Masking requires $O(N^2)$ SMR calculations using the algorithm described at the beginning of this chapter followed by $O(N\log N)$ operations to sort the unmasked partials. If few partials are masked, both operations approach their asymptotic costs. Although we initially hypothesized that the masking strategy would be too costly for the expected increase in reduction quality, the fact that the amplitude and masking strategies showed nearly identical results leads us to choose the amplitude-based reduction strategy.

The results suggest that at least half the partials can be pruned from sinusoidal or resonance models and in some cases more partials can be pruned. However, the question remains how many partials can be removed from a model? Requiring that users perform similar experiments on all their sound models is not a practical solution. Moreover, synthesis in real-time music applications is often based on the result of musician-controlled transformations which can quickly and dramatically change the components of a model.

One method for estimating the reducibility of a model is the sum-of-amplitudes ratio described in equation 4.1. As stated above, the results suggest that each frame of a sinusoidal model can be reduced without significant degradation until the sum of amplitudes

in the reduced frame is less than three quarters of the sum of amplitudes in the original frame. If this threshold is lowered to between one half and two thirds, the reduced models exhibit moderate degradation (i.e., average listener scores between 3 and 4). We therefore propose the following real-time sinusoidal-model reduction algorithm:

```
REDUCE-SINUSOIDS(model, n)
    frame ← the frame in model at time n
    N ← number of partials in frame
    A_N ← Σ_{i=1}^{N} A_i(n), where A_i is the amplitude of the ith partial in frame
    A_R ← 0
    reducedframe ← {}
    while A_R < (3/4) A_N
        do max ← the largest partial remaining in the original frame
           A_R ← A_R + A_max
           Add partialmax to reducedframe
           Remove max from the original frame
    return reducedframe
```

This algorithm requires N steps to compute the total amplitude and at most $N - 1$ steps to construct the reduced frame.

In order to improve the quality of aggressive reductions of sinusoidal models, the issue of frequent births and deaths of partials must be addressed. Garcia and Pampin suggest discarding sinusoidal tracks shorter than a specified minimum if their average SMR measured over several frames is below a specified threshold [51]. In a real-time application with mutable sounds, we can only look at tracks backward in time. Moreover, we do not want to calculate averages over a large number of tracks because this approach increases computational cost and reduces temporal accuracy (e.g., the effect of a dramatic change in user input may be "smeared" over several frames in the reduced model because of averaging). A heuristic that works surprisingly well on test sounds is to keep the partials in order of increasing frequency. Most natural sounds have a spectrum in which the amplitudes of spectral peaks decrease as frequency increases. For models of these sounds, REDUCE-SINUSOIDS still retains the partials with the strongest amplitudes but also retains a small

number of weaker partials of low frequency. Because these partials fluctuate in amplitude between frames, they are sometimes pruned and sometimes maintained when frames are sorted by amplitude before reduction. Preserving these "middle-strength" partials, particularly in the 500-5000Hz frequency range, increases the number of partials in the reduced model, but reduces the perception of birth and death events. In addition, computation time is saved by avoiding the sorting step. Some models, such as models of vocal sounds in which the amplitude increases at higher frequencies (i.e., near formants) or sounds with noisy or flat spectra, may lose salient partials using this heuristic, although they will still benefit from the decrease in births and deaths. For this reason, sorting frames by amplitude is retained as an option for the algorithm.

Births and deaths are not an issue for resonance models, although we do want to modify the sinusoidal algorithm to include another temporal property, the decay rate. The results of the resonance model experiments emphasized the importance of partials with longer decay rates, particular in the bass and tam-tam models. Instead of using the initial amplitude of a resonance partial, we can compute its theoretical amplitude contribution over the duration of the sound as the ratio of initial amplitude to bandwidth:

$$\widehat{A}_i = \int_t^\infty A_i e^{-\pi k_i t} = \frac{A_i}{\pi k_i} \tag{4.2}$$

Substituting this formula for initial amplitude, we estimate that the sum of contributions (i.e., $\widehat{A}_i$ values) in a reduced model must be at least 90% of the original sum to maintain high quality. Although this sounds like a conservative estimate, it actually corresponds to a reduction by 7/8 for the marimba and by one half for the bass and tam-tam. The reduction algorithm for resonance models is as follows:

REDUCE-RESONANCES(*model*)
  $N \leftarrow$ number of partials in *model*
  $\widehat{A}_N \leftarrow \sum_{i=1}^{N} \widehat{A}_i$
  Sort partials in *model* by decreasing values of contributions $\widehat{A}_i$
  $A_R \leftarrow 0$
  reducedmodel $\leftarrow \{\}$

while $\widehat{A}_R < \frac{9}{10}\widehat{A}_N$
  do $max \leftarrow$ the largest partial remaining in the original frame
    $\widehat{A}_R \leftarrow \widehat{A}_R + \widehat{A}_{max}$
    Add *partialmax* to *reducedmodel*
    Remove *max* from *model*
return *reducedmodel*

Note that the resonance-reduction algorithm is not a function of time and is employed only when the input resonance model has been modified.

Chapter 5

Perceptual Scheduling

This chapter describes the perceptual scheduling framework for dynamically trading audio quality for reduced computational bandwidth to prevent QoS failures. Perceptual scheduling is first described as a general algorithm that uses perceptual constraints. It is then used in conjunction with the sinusoidal-model and resonance-model reduction strategies developed in the previous chapter to reduce the computational demands of synthesis algorithms. Run-time performance for several examples is measured to determine the effectiveness of both the scheduling framework and reduction strategies.

5.1 Preventing QoS failures

Chapter 3 discussed a potential QoS failure in which the time to compute a sample exceeds the sample period (i.e., about 20μs if the sampling rate 44100Hz). If this bound is exceeded, the music synthesis system will be unable to maintain the required number of samples in the audio output buffer. This buffer underflow may cause audible gaps or clicks in the sound output and increase the latency and jitter between controller input and audio output.

To prevent such failures, it is necessary to provide a feedback signal to the running patch, as illustrated in figure 5.1. The feedback can then be used by transforms in the patch

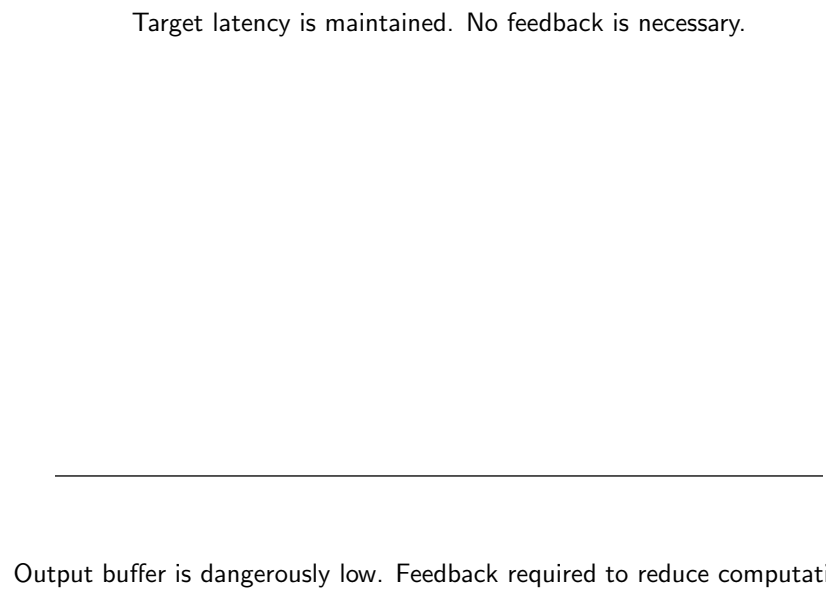

Figure 5.1: When the real-time guarantees are satisfied, the perceptual scheduler does nothing. When the process fails to meet the guarantees, the perceptual scheduler provides feedback to the program to reduce its computation.

to reduce their computational requirements and lower the execution time below the real-time bound. Transforms that reduce their computational requirements must do so in a way that maintains the quality of the sound output as the execution time decreases. The problem of balancing sound quality and computational bandwidth is called *perceptual scheduling*.

5.2 Generic Perceptual Scheduling Problem

Bounded execution time is maintained in perceptual scheduling by dynamically reducing the computational requirements of certain transforms, called *reducible transforms* when the measured execution time exceeds the bound. Reducible transforms generate sound models or synthesize waveform representations. Given a set of reducible transforms R, the measured per-sample execution time E and an upper bound on execution time E_{max}, the scheduler executes the following algorithm once per epoch:

```
PERCEPTUAL-SCHEDULER(E, E_max, R)
  if E > E_max
    then for each r ∈ R
           do calculate c(r),the estimated computation saved by reducing r
         find R' ⊂ R such that E − Σ_{r∈R'} c(r) ≤ E_max
         for each r ∈ R'
           do switch r to reduced-computation mode
    else for each r ∈ R
           do if r is executing in reduced-computation mode
                then allow r to incrementally increase computation
```

During the next epoch, all reducible transforms in the selected subset R' will be executed in reduced-computation mode. If the measured execution time is less than the bound, the algorithm allows reduced transforms to increase the computation incrementally over several epochs so that they do not become "stuck" with a computational bandwidth that no longer maintains sound quality as the model changes over time. If the gradual increases cause the execution time to once again exceed the bound, the algorithm will select a new subset of transforms to reduce.

It should be noted that the computational savings estimate $c(r)$ may not reflect the computation saved only by transform r, but also the computation saved by transforms that depend on the result of r. For example, reducing the size of the output from a transform that generates a model (or reads a model from disk) will reduce the computational bandwidth of a transform that synthesizes a waveform representation from the model. In fact, the

process of reducing the model size may actually increase the computation required by the generating transform, but if there is a greater decrease in the computation required for synthesis, the net computational bandwidth will be reduced.

There are several possible strategies for choosing R'. One is to find a subset that minimizes the amount of computational reduction but still brings the total execution time below the bound. The problem can be stated as follows:

$$\begin{aligned} Minimize \quad & \sum_{r \in R'} c(r) \\ subject\,to \quad & E - \sum_{r \in R'} c(r) \leq E_{max} \end{aligned}$$

This optimization can be solved using 0-1 integer programming. For each transform r, define a variable s_r that is 1 if $r \in R'$ and 0 if $r \notin R'$. The problem can then be restated:

$$\begin{aligned} Minimize \quad & \sum_{r \in R} c(r) s_r \\ subject\,to \quad & E - \sum_{r \in R} c(r) s_r \leq E_{max} \end{aligned}$$

An exhaustive search of the problem space takes $O(2^{|R|})$ steps. However, if a particular subset S fails the constraint, then all subsets of S will also fail, which means $2^{|S|}$ elements can be pruned from the problem space. In the corresponding integer programming problem, if a particular solution fails, switching any of the variables in the solution from 1 to 0 will also fail. If few subsets satisfy the constraint then the size of problem space will be greatly reduced by pruning. In fact, if the constraint fails for the entire set R, the algorithm fails immediately. However, if many small subsets satisfy the constraint, the algorithm must search them all to find the minimum.

Another approach to pruning the search space is to grow the search space incrementally. Starting with a space S containing a single transform $\{r\}$, transforms are added until a subset is found that satisfies the constraint.

```
FIND-REDUCED-SUBSET(S)
  result ← TEST-SUBSETS(S)
  if result ≠ fail
    then return result
    else if there exists a reducible transform r ∉ S
           then return FIND-REDUCED-SUBSET(S ∪ r)
           else return fail

TEST-SUBSETS(S)
  if E − Σ_{r∈R'} c(r) ≤ E_max
    then for each S' ⊂ S
           do result ← TEST-SUBSETS(S')
              if result ≠ fail
                then return result
         return S
    else return fail
```

FIND-REDUCED-SUBSET is initially called with a single transform. This algorithm has a worst-case running time of $O(2^{|R|})$ and may not find the optimal subset (i.e., the subset with the smallest reduction in computational bandwidth). However, it does favor smaller subsets and only requires computational-savings estimates for the transforms that are searched before finding a solution. Solutions with fewer transforms are usually better because the model-reduction algorithms can be expensive. Reducing the number of estimates required is significant because estimating the savings from a transform may also require an expensive analysis of its sound model. If no solution exists, the algorithm will fail in $O(|R|)$ steps.

Each step of FIND-REDUCED-SUBSET exhaustively searches all subsets of S. If only S and the most recently added transform are tested, a solution can be found in $O(|R|)$ steps:

```
BUILD-REDUCED-SUBSET-LINEAR(R')
  if E − Σ_{r∈R'} c(r) ≤ E_max
    then return R'
    else if there exists a reducible transform r ∉ R'
           then if E − c(r) ≤ E_max
                  then return {r}
                  else return BUILD-REDUCED-SUBSET-LINEAR(R' ∪ {r})
           else return fail
```

This algorithm will find a solution in $O(|R|)$ steps if it exists and fail after exactly $2|R|-1$ steps otherwise. It will favor a solution using only one transform if it exists. However, it ignores most of the intermediate-sized subsets. It may recommend that all the transforms be reduced even if a smaller non-single subset exists that satisfies the constraints. Despite returning suboptimal results in some situations, this algorithm is preferred for real-time use because bounded execution time is more important than minimal execution time. An optimal solution may result in less computation within an epoch, but the more complex algorithm will increase the computation at epoch boundaries, exacerbating the potential QoS failures that triggered the scheduling algorithm in the first place.

5.2.1 Choosing the epoch length

The choice for a scheduling epoch must balance the cost of the scheduling algorithm with the need for responsiveness. If the epoch is too long, the algorithm will not be able to respond to changes in the sound models used by reducible transforms. Such changes may affect the computational bandwidth required to maintain sound quality and choice of a reducible subset. If the epoch is too short, the overhead of scheduling will be greater than the computation saved from the reductions.

For music based on note events (i.e., most traditional and contemporary music), the sound model may be updated when a new note begins. The frequencies and amplitudes can be scaled according to the desired pitch and loudness of the new note. The timbre of "physically-inspired" models may also be adjusted to reflect natural timbral changes between notes. Iyer reports temporal resolutions of approximately 300-800ms for beats, 75-200ms for subdivisions of beats (e.g., sixteenth notes) and $\pm$30ms for expressive-timing deviations between the prescribed length of a note and its actual length (i.e., a sixteenth note at a tempo of 120bps should last 125ms, but in an expressive performance may be as long as 155ms or as short as 95ms depending on its musical context) [57]. A study by

Clarke reports a sensitivity of 20ms to simple isochronous sequences (e.g., evenly-spaced clicks), although this sensitivity dropped to 50ms in melodic examples [27]. Sensitivity also decreases at the beginning and end of phrases in traditional Western music [98] and during temporal jitter, or "slop" in rhythmic patters [58].

Timbre can also change within a note event (e.g., bowing on a string instrument or adjusting breath and embouchure on a wind instrument), and some electronic works eschew note events in favor of continuous timbral changes (e.g., Wessel's *Antony* and Chowning's early frequency-modulation compositions [24]). While the timing sensitivity of 10 ± 1ms reported in chapter 3 applies to a performer using continuous control, the listener has very little ability to detect the exact timing of changes within a continuous motion [99]. It should also be noted that a computationally reduced timbre is not necessarily "frozen" during an epoch. The timbre can change at the reactive resolution of the system (i.e., the clock period described in chapter 3), but the computational bandwidth must remain within the reduced bounds set by the scheduler.

With these considerations in mind, the default epoch of the perceptual scheduler is set at 30ms, with the option of increasing or decreasing the length in individual applications.

5.3 Additive Synthesis

For applications using additive synthesis, perceptual scheduling minimizes the number of partials synthesized with respect to constraints that preserve the quality of the sounds being modeled. Typically, one reducible transform is required per model being used. The computational savings estimate for each model is the number of partials to be pruned multiplied by the per-partial computational cost of the synthesis algorithm. Using the per-partial cost of 0.053μs per sample per partial for oscillator-based additive synthesis (i.e., as reported in section 3.1.1), the estimated computational savings after pruning 100 partials from a model is 5.3μs per sample. If TDAS is used for additive synthesis, the estimated savings is 1.6μs per sample (i.e., using the cost of 0.0016μs per sample per partial

reported in section 3.1.2).

The following subsections describe implementation of a reducible transform that use the sinusoidal-model reduction strategy described in chapter 4, evaluate its performance on different models, and propose the use of customized reducible transforms that have better performance given additional knowledge about sound models.

5.3.1 Implementation

The sinusoidal-model reduction algorithm REDUCE-SINUSOIDS developed in chapter 4 has been incorporated into the OSW transform sdif::ToReducedSinusoids. ToReducedSinusoids is a modification of the transform sdif::ToSinusoids that applies the reduction algorithm to models as they are read from SDIF files. The output of this transform can then be used for synthesis, as illustrated in 5.2a. For models that are not read from SDIF files, but generated algorithmically (e.g., the band-limited sawtooth model used in chapter 3), an additional transform ReduceSinusoids is included that applies the reduction algorithm to a sinusoidal model input. An example using ReduceSinusoids is shown in figure 5.2b.

Synthesis servers that use the reduction algorithms must also include the transform PerceptualScheduler. This "global transform" interacts with the system to implement the perceptual scheduling algorithm. It is not connected to other transforms in the patch because it does not directly process any signal or control data.

5.3.2 Performance

The real-time performance of the reduction strategies was measured on several sound models. In addition to the performance of models from stored SDIF files that used sdif::ToReducedSinusoids for reduction, the performance of a 100Hz band-limited sawtooth model frequency-modulated by a 4Hz oscillator (i.e., the "LFO" function found in many analog synthesizers) using ReduceSinusoids was also measured. Measurements using both sinusoidal oscillators and TDAS were taken, first using no perceptual scheduling or reductions

a)

b)

Figure 5.2: Synthesis servers with reducible transforms. a) sdif::ToReducedSinusoids reduces a model read from the SDIF buffer mymodel before it is synthesized. b) ReduceSinusoids is used to reduce the sawtooth model generated by AnalogVCO. In both examples, the global transform PerceptualScheduler manages the reducible transforms.

Figure 5.3: Performance comparison oscillator-based synthesis on full and reduced versions of the "suling" model. The per-sample CPU time is calculated each epoch over the duration of the model.

---

and then with the scheduling and reduction algorithms engaged. Figure 5.3 compares CPU usage, measured in microseconds per sample, over time for oscillator-based synthesis (i.e., AddSynth) of the full and reduced versions of the suling model used in chapter 4. Figure 5.4 compares synthesis of full and reduced versions of the suling model using TDAS (i.e., AddByIFFT). In both cases, the scheduling and reduction algorithms consistently reduced the CPU time over the entire synthesis process. Equally important are the tighter upper and lower bounds on CPU usage when the scheduling and reduction algorithms are used. The reduced streams are bounded, while CPU usage in the full streams grows. The spikes at 0.5 seconds in the reduced streams 0.05 seconds in the full streams are startup anomalies

Figure 5.4: Performance comparison of TDAS on the suling model.

---

that are model-dependent. CPU times are bounded between 2 and 2.5 microseconds per sample when AddSynth is used, and between 2.5 and 2.7 microseconds per sample when AddByIFFT was used. The CPU times are more varied when perceptual scheduling was not used. The wider variation can be attributed mostly to the change in the number of partials in the full model over time. Thus, the dependency of execution time on data size has been reduced.

Similar results were observed for other models. Figure 5.5 shows summary results for the suling model, the modulated-sawtooth model, and three additional models based on a phrase sung by Khyal singer Shafqat Ali Khan [129], a brief passage performed by saxophonist Steve Coleman, and a recording of an "angry cat" used in the analysis-synthesis

Figure 5.5: Comparison of average CPU usage between synthesis of full and reduced versions of sinusoidal models. The upper graph shows the results when AddSynth is used while the lower graph represents the results when AddByIFFT is used.

Figure 5.6: Comparison of generic and customized reduction strategies for a frequency-modulated 100Hz band-limited sawtooth model. The graph displays the average CPU time for the full model and equal-sized reductions using both reduction strategies. Results using both synthesis algorithms are included.

---

panel at ICMC 2000 [133]. Most of the models show a strong decrease in computational bandwidth because most of the energy is concentrated in a few partials. A greater decrease in computation is observed for larger models because the overhead of the synthesis, scheduling and reduction algorithms is constant and therefore a smaller percentage of the total per-sample CPU time.

5.3.3 Customized Reduction Strategies

More efficient reduction strategies can be developed given additional knowledge about the sound models. For example, the band-limited sawtooth model contains sinusoids

with $1f, 2f, 3f, \ldots, Nf$ and amplitudes $1, 1/2, 1/3, \ldots, 1/N$, where f is the pitch and $N < S/2f$. The sum of amplitudes in the model is the harmonic series of length N:

$$H_N = \sum_{i=1}^{N} \frac{1}{i} \approx \ln(N) + \gamma \tag{5.1}$$

where $\gamma \approx 0.5772156649$. Using the heuristic from chapter 4 that the sum of amplitudes in the reduced model should be at least three quarters of the sum of amplitudes in the original model, we can solve for N_R, the number of partials in the reduced model:

$$\ln N_R + \gamma \approx \frac{3}{4}(\ln N + \gamma) \tag{5.2}$$

$$N_R = \lfloor e^{\frac{3}{4}(\ln N + \gamma)} - \gamma \rfloor \tag{5.3}$$

Substituting $\lfloor S/2R \rfloor = \lfloor 44100/(2 \cdot 100) \rfloor = 220$ for N, the reduced 100Hz sawtooth model used in the previous section should contain 49 partials, which is also the number of partials remaining when the generic reduction strategy REDUCE-SINUSOIDS is used. However, unlike the generic reduction strategy, which requires $\theta(N)$ operations to determine the size of the reduced model, this strategy computes the size in constant time. Moreover, the size can be computed without first examining the partials in the model, so the reduced model can be generated directly.

Although there is a significant reduction in complexity, a comparison of CPU usage by the generic and customized strategies shows a decrease of about one microsecond per sample when using the customized strategy with either AddSynth or AddByIFFT, as illustrated in 5.6. A more complex customized reduction strategy will be evaluated on a musical example in chapter 6.

5.4 Resonance Modeling

Perceptual scheduling of resonance models minimizes the number of partials synthesized, with one reducible transform per resonance model and a computational savings estimate proportional to the number of partials pruned from each model. Using the per-partial cost of 0.047μs per sample per partial for resonant filterbank synthesis, the estimated

Figure 5.7: A reducible resonance-model server. A model read from the SDIF buffer mymodel is reduced using ReduceResonances when computation reduction is required. The resonant filterbank Resonators is updated whenever ReduceResonances changes the size the model.

---

computational savings after pruning 100 partials is 4.7μs per sample. The remainder of the section describes the implementation of a reducible transform for resonance models and evaluates in performance.

5.4.1 Implementation

OSW includes a transform ReduceResonances that applies the reduction algorithm Reduce-Resonances to resonance-model representations. Because resonance models are constant over time and they are represented in SDIF by a single frame of constants that is read once before the model is used, no significant performance is gained by a separate

transform that reduces a model as it is read from an SDIF representation. However, this observation does not mean the reduction algorithm is applied only once to the model when it is read. The reduction algorithm must be applied each scheduling epoch because the need for reduction may vary as the performance of other transforms changes. A patch using ReduceResonances is illustrated in figure 5.7. Once again, a PerceptualScheduler transform is required to use the reduction algorithms.

To reduce computation in resonant filterbank synthesis, it is also necessary to remove filters from the bank that correspond to pruned partials in the reduced model. Otherwise the "orphaned" filters will continue to operate and no computation is saved. The Resonators transform was modified to expand and contract the filterbank dynamically as the size of the model changes.

5.4.2 Performance

The performance of the resonance-model reduction strategy was measured on the marimba, tam-tam and bass models used in chapter 4 as well as a model of a low A (i.e., 55Hz) on a piano. In each test, an impulse was sent to Resonators once every second for 10 seconds. The CPU usage over time is plotted for the tam-tam models in figure 5.8. A performance summary for all four models is provided in figure 5.9. Similar to the sinusoidal-model results, all examples consistently used less CPU time with tighter bounds when the scheduling and reduction algorithms were applied. In fact, the piano model, which contained 697 partials, required an average of 34.1 microseconds of CPU time per sample and therefore did not run in real time on the reference machine. When the reduction algorithm was applied, the piano model required an average of only 11.3 microseconds per sample, making its synthesis possible in real time.

The performance of the resonance-model reduction strategy was only measured with impulse excitations. As discussed in chapter 3, the computational complexity of a resonant filterbank is not dependent on the excitation signal. However, it is not clear

Figure 5.8: Performance comparison of full and reduced versions of the tam-tam model. The per-sample CPU time is calculated each over a 10 second period.

---

that the reduction strategy used for impulse-driven models will work for models with other excitation signals. The design and evaluation of reduction strategies for resonant filterbanks with non-impulse excitations are left as open questions for future research.

5.5 Discussion

The performance evaluations presented in this chapter show that perceptual scheduling combined with reduction strategies is effective at reducing the computational requirements for synthesizing individual sinusoidal and resonance models. A perceptual scheduling system maintains strongly bounded execution times, which are necessary to maintain real-time QoS guarantees. The next chapter will evaluate the performance of a perceptual

Figure 5.9: Comparison of average CPU usage between synthesis of full and reduced versions of resonance models.

---

scheduling system on multiple models in larger musical examples.

In larger examples with more than one model, the option exists to combine all the models and apply the appropriate reduction strategy once on the aggregate model. This approach increases the likelihood of finding an optimal reduction that prunes partials according to their salience in the aggregate model. However, it is unclear whether additional partials can be pruned from the aggregate without using the $O(N^2)$ masking strategy, where N is the total number of partials in all of the models. Per-voice reductions also offer more flexibility, including the labeling of some models as "non-reducible" and selection of different reduction strategies for different models.

Chapter 6

Evaluation of Perceptually Scheduled Music

This chapter describes the evaluation of perceptual scheduling on larger musical examples. The evaluation includes analyses of run-time performance (i.e. usage of CPU time) and subjective rating by human listeners in a controlled experiment.

Table 6.1 lists the musical examples used in the evaluation. The table includes the synthesis algorithms used, the number of partials in the sound representations and the number of "voices" (i.e., representations synthesized simultaneously) used in each example. The musical examples were chosen to use different algorithms (e.g., oscillators, TDAS and resonant filterbanks) and numbers of reducible transforms in order to evaluate perceptual scheduling under different computational requirements. Additionally, the musical examples are drawn from a wide range of musical sources and disciplines. The Bach fugue and the excerpts from *Constellation* use a traditional note-based performance model in which the synthesis models are scaled and resynthesized for each note event. On the other hand, the Tibetan-recording improvisation and *Antony* employ continuous control of the models over the duration of the music. There is no true separation into discrete "notes." While the two note-based works are specified in scores, the works using continuous control are

Name / Description	Duration	Synthesis	Partials	Voices
Fugue in B♭ minor, J.S. Bach	34s	TDAS	650	5
Improvisation on Tibetan Recording	28s	Oscillators	202	2
Antony, D. Wessel	45s	TDAS	600	3
Constellation, R.B. Smith (1)	6s	Resonances	288	6
Constellation, R.B. Smith (2)	21s	Resonances	273	2

Table 6.1: Summary of musical examples, with durations, synthesis algorithms, number of partials and number of voices.

---

improvised within specified constraints. In one example, the twentieth-century technique *musique concréte*, or music from recorded material, is applied to traditional Tibetan vocal music.

This chapter is organized as follows. Section 6.1 describes the experimental methods used in the listening tests. The results of both performance analysis and listener evaluation are then presented separately for each of the musical examples in sections 6.2 through 6.5. Section 6.6 concludes the chapter with a more general discussion of the results.

6.1 Experimental Methods

A group of 15 experienced listeners participated in an experiment that measured the effects of perceptual scheduling on sound quality. Listeners were presented with pairs of sounds in which one was identified as the original and one as the reduced-computation version. Similar to the experiments described in chapter 4, the listeners were asked to rate any perceived difference on a scale from 1 to 5, as illustrated in table 6.2.

Listeners were presented versions of each sound example with full computation, maximally reduced computation (i.e., performing all allowed reductions within the constraints set by the algorithms) and a partially reduced computation (i.e., a subset of the reductions were performed to meet a specified upper bound on CPU time). The exception was the Bach fugue, for which only two versions with full computation and fully reduced compu-

Score	Relative Quality	Comparison to original
5	Excellent	Virtually indistinguishable
4	Good	Just perceptibly different
3	Fair	Degraded, but adequate
2	Poor	Strongly degraded
1	Unsatisfactory	Barely comparable

Table 6.2: Scoring system for listening experiments. The scoring system is the same as used in chapter 4.

---

tation were presented. Each version was presented to the listener with the full-computation version, for a total of 14 example pairs. For each example, the full-computation version was presented once with itself as a control. The 14 example pairs were then presented to the user in a random order. The panelists were free to listen to the two sound samples in each pair in any order as many times as they wanted. Additionally, because the sound samples were longer than in the previous experiments (e.g., 34 seconds and 45 seconds for the Bach-fugue and *Antony* examples, respectively), listeners were given a "scrub" control to fast-forward and rewind to arbitrary points within the sound sample after listening to it at least once from start to finish. This extra degree of freedom allowed listeners to focus on brief segments in which they heard perceived differences.

The experiment was conducted at CNMAT during the spring of 2001. All sessions occurred in the recording studio under uniform listening conditions (e.g., speaker levels were calibrated to an 80dB level for a 1000Hz test tone), and used an automated interface similar to the previous experiments, as described in section 4.1.

Because the reduction strategies were designed to minimize degradation, the perceptual differences between original and reduced sound examples were more subtle than those in the experiments reported in chapter 4. After completing the experiment, listeners were invited to comment on the experiences. Feedback from listeners revealed two distinct approaches used by listeners to rate the quality. The majority of listeners considered the examples in their entirety, using readily perceived artifacts as indicators of degraded qual-

ity. When such artifacts were observed, they used the scrub control to focus on location of those artifacts. Other listeners assumed that artifacts must exist and used the scrub control to locate artifacts that they did not hear when listening to the full sample. Although this listening strategy is equally valid given the guidelines of the experiment, listeners who used it usually gave lower scores to the examples that included reductions. (In fact, they were also more likely to give lower scores to examples that did not use any reductions!)

In addition to the listening experiment, the musical examples were evaluated by an outside panel of experts in sound analysis and synthesis, including Professors Lippold Haken from the University of Illinois and Kelly Fitz from the University of Washington. Their feedback was used to guide the analysis of results.

6.2 Example 1: *Fugue in B♭ minor*, J. S. Bach (BWV 867)

In the first musical example, measures 10 through 25 of Fugue 22 in B♭ minor (BWV.867) from the *Well Tempered Clavier, Book I* by Johann Sabastian Bach was performed using a sinusoidal model of a harpsichord [12]. This particular fugue was chosen because it uses five voices, which is an unusually large number for a fugue. The excerpt begins as the third, fourth and fifth statements of the subject enter, as illustrated in figure 6.1, and concludes at a major cadence.

A MIDI sequence from the Classical MIDI Archive at `http://www.prs.net` was used as the "performer" [100]. The notes from the MIDI file were separated by fugue voice and sent to separate subpatches, each of which contained a separate instance of the harpsichord model and AddByIFFT synthesis transform. When one of the subpatches received a new pitch value from the MIDI file via its pitch inlet, the harpsichord model was scaled to the new pitch and restarted from the beginning by setting the connected time machine to zero. The synthesized waveform output from the AddByIFFT transform was mixed with the other voices.

The harpsichord model used in each voice contained 130 partials, so 650 partials

Figure 6.1: Measures 10 through 16 of Fugue 22 from the *Well Tempered Clavier*. The voices are shown on separate staves for clarity.

Figure 6.2: CPU usage during full and reduced performances of the Fugue 22 excerpt. Because the two streams overlap frequently, they are displayed on separate graphs. More computation is used when more notes are played simultaneously. However, the reduced version mostly remains below the real-time constraint of 20μs per sample.

were used by all five voices. In addition to the computation required to synthesize the partials using five instances of AddByIFFT, an additional 650 frequency-scaling operations per clock period were required to assign the correct pitches to the models. Without perceptual scheduling enabled, the patch did not run in real time on the reference machine, as illustrated in figure 6.2. The large fluctuations in CPU usage over time apparent in figure 6.2 correspond to the changing number of voices active in the music (i.e., the *texture*). When notes sound simultaneously in several voices, more computation is required. For example, the general increase over the first twelve seconds (i.e., the increasing height of the plateaus in the graph) corresponds the entrances of the fourth and fifth voices in measures 12 and 15, respectively. When perceptual scheduling was enabled and the computational bandwidth was bounded at 20μs per sample (i.e., 100% CPU usage on the reference machine), the computation saved by reducing all five voices allowed the patch to run in real time. The computation still changes with the texture of the music (i.e., more computation when more voices sound simultaneously), but mostly within the specified constraints.

The results of listener evaluations of the fugue performance under full-computation and reduced-computation conditions are illustrated in figure 6.3. The position along the horizontal axis represents the average CPU times of the full and reduced performances, which were 22μs/sample and 17μs/sample, respectively. The large error bars along the CPU-time axis represent the large changes in CPU usage due to musical texture, as described above. The vertical axis represents the average listener scores on the 5-to-1 scale described in section 6.1. The harpsichord model was similar to the single-tone models used in the experiments described in chapter 4. Because the reduction strategies were refined using single-tone models, we expected the listener scores to remain high (i.e., between four and five). Although listeners observed some degradation, particularly during the attack transients of each note, most gave the reduced sample a score of three or four, depending on how serious they considered the artifacts. Fitz observed a small but noticeable decrease in brightness and spectral roughness in addition to the artifacts in the attack transients,

Figure 6.3: Quality versus average CPU time in Fugue 22. Quality is reported as the mean listener scores between 1 and 5. The original (i.e., full-computation) performance is represented by the round point, while the reduced-computation performance is represented by the square point. The horizontal error bars represent the deviation in CPU time over the duration of the example. The vertical error bars represent the standard deviation in listener scores.

---

but nonetheless thought the sample deserved a score of four [41]. Haken considered the artifacts more serious, and recommended a score of two [54]. Some listeners considered both the original and the the reduced samples degraded and gave them consistently lower scores. Additionally, listeners who had extensive experience with traditional Western music were expected to be more critical of this example. However, the mean listener-score remains close to four, and we consider this example a successful instance of graceful computational reduction.

6.3 Example 2: Time-scale Improvisation on Recording of Tibetan Singing

In the previous example, a model of an acoustic instrument was used to realize the notes of a piece of music originally intended for the instrument being modeled. This example uses sinusoidal modeling for music that could not have been realized acoustically. As described in chapter 2, sinusoidal models can be used to produce new music by dynamically changing the time function as the model is synthesized. In this example, a sinusoidal model made from a recording of Tibetan singer Tsering Wangmo in 1996 was used. The model was synthesized simultaneously on two separate voices using AddSynth transforms for oscillator-based additive synthesis and output over separate audio channels, as illustrated in figure 6.4. Although two synthesizers were used, both shared a single SDIFBuffer resource for the model. One of the two models was connected to a TimeMachine transform, which was used to dynamically scale the time function of the sinusoidal model, allowing it to be sped up, slowed down or performed backwards according to user input. The original and time-scaled versions of the model were synthesized simultaneously, thus creating an improvised "counterpoint" of the original against its time-scaled variation. Because the Tibetan passage uses a pentatonic scale, any timed-scaled or retrograde variations still sound consonant when played against the original.

The patch was performed by volunteer improvisors using a mouse to vary the rate of the time machine between -2 and 3 (i.e., backwards at twice the speed and forwards at three times the speed). All mouse-controlled rate changes were captured as time-stamped events that could be played back later. One of the captured improvisations was selected for the computational and listener-evaluation study.

The CPU usage of the full-computation and two reduced-computation performances of the selected improvisation is illustrated in figure 6.5. In the fully reduced version, the upper bound was set at a low level so that the scheduler would reduce both voices at

Figure 6.4: OSW patch for Tibetan-recording improvisation. The two synthesizers share a single SDIFBuffer resource containing the sinusoidal model, but have separate virtual time sources and output samples to separate audio channels. The synthesizer on the right is connected to a time machine for user control of virtual time.

---

all times. In the partial reduction, the upper bound was set at 10μs per sample. The sections of the partially reduced stream that have greater average CPU times than the fully reduced stream represent epochs during which only one of the two voices was reduced in order to meet the upper bound. The spikes represent frames of the model from which few partials were reduced and less computation saved. Such frames, which are associated with triansients or low-amplitude sections of the model, occur at infrequent intervals when the model is played at full speed from beginning to end, but can be stretched out over longer periods when virtual time is slowed down. More computation is also required when virtual time is sped up because birth and death events must be handled with greater frequency.

Figure 6.5: CPU usage during full and reduced performances of the Tibetan-recording improvisation.

Changes in virtual time rate also require that the SDIF frames not be read in ascending order, but rather the entire model searched to find the closest match in time, which increases the computation necessary to synthesize a frame. In fact, the increases in CPU time in the intervals between 16 and 17 seconds and between 21 and 22 seconds correspond to periods in the improvisation where the virtual time rate is double the physical time rate or the rate is changing rapidly. Additionally, the 10μs/sample upper bound of the partially reduced version was exceeded during the interval between 20 and 22 seconds. This suggests that the computational cost, which does not account for the frame search algorithm, was underestimated by the perceptual scheduling algorithm.

As illustrated in figure 6.6, this example fared very poorly under perceptual

Figure 6.6: Quality versus average CPU time in the Tibetan-recording improvisation. Quality is reported as mean listener scores between 1 and 5. The full-computation performance is represented by the round point, while the triangular and square points represent the partially reduced and fully reduced performances, respectively. The reduced examples received poor scores from listeners.

---

scheduling. All listeners gave the reduced samples scores of three or less, indicating that the samples were still recognizable but severely degraded when compared to the original. Most of the degradation can be attributed to birth and death artifacts, particularly during transitions and vibrato. During these events, the amplitudes of the partials change rapidly between successive frames, so the sizes of the reduced models change rapidly as well. While some birth and death artifacts are masked in models with broader spectra and more noise-like features, such as the James Brown example, little masking of artifacts occurs in the more harmonic vocal models. Haken's reduction strategy, which uses a global

maximum amplitude value for each partial instead of different amplitude values on successive frames, might be expected to do better on this example [53]. Similarly, Garcia and Pampin's strategy that prunes short partials might avoid selection of shorter partials in the reduced model [51]. However, as stated in section 4.3, these strategies do not account for users' changing the model in real time, as is done in this example. Fitz's reaction to the reduced samples was that they sounded like the results of a sinusoidal analysis where the user erroneously leaves the analysis parameters (e.g., spectral resolution, window size and minimum pitch) unchanged from their defaults [41]. Building on this observation, the poor results in this example suggest that one reduction strategy cannot fit all MQ-style additive synthesis models. Additional reduction strategies will need to be developed with heuristics for different classes of sinusoidal models, such as human voices. However, like the generic strategy for single-tone models, a heuristic strategy should not rely on knowledge about the future of a model. In the following example, such a customized strategy will be evaluated.

Another option is adding a parameter to increase the number of partials retained, sacrificing some computational savings for increased quality. Retaining more partials decreases the likelihood of birth and death artifacts. Although this solution allows a single parameterized algorithm to be used with more models, the added degree of freedom can make its integration into real musical applications more difficult. As with the parameterized analysis/synthesis algorithms, use will require additional expertise.

6.4 Example 3: *Antony*, David Wessel

This example is based on *Antony*, originally composed by David Wessel in 1977 [127] and realized on a digital oscillator bank designed at IRCAM by G. DiGiugno called the 4A [37]. This version of *Antony* uses three voices containing 200 sinusoids each. It begins with the sinusoids in each voice distributed uniformly on a log-frequency scale between 32.5Hz and 5kHz. The piece then unfolds over a series of epochs during which the sinusoids in each voice "migrate" to frequency regions specified by the user in real time according to

the following algorithm:

```
ANTONY-VOICE-MIGRATION(f_lo, f_hi, rate)
   period ← 1/rate
   for i ← 1 to N
      do sinusoids[i].frequency = f_lo + random() * (f_hi − f_lo)
         sinusoids[i].amplitude = ampscale[sinusoids[i].frequency]
          Wait period seconds
```

where *sinusoids* is the model, N is number of sinusoids in the model (i.e., 200), *rate* is the migration rate, and f_{lo} and f_{hi} are the new upper and lower frequency bounds, respectively. When the algorithm ends, all the sinusoids have a random frequency within the new bounds. The algorithm executes concurrently for all voices during an epoch. The epoch ends after the algorithms terminates. At the beginning of the next epoch, the algorithms are restarted with the new values of f_{lo}, f_{hi} and *rate* for each voice. The array *ampscale* is a table that scales the amplitude of each sinusoid as a function of its frequency. It uses a quadratic function to model the equal-loudness contours that normalize perceived loudness across different frequency bands [1].

The effect of synthesizing audio from this process is, in the words of the composer, motion "between vague and undifferentiated clusters and clear pitches." When the bounds are far apart, the effect is a dense, noise-like sound with no determinate pitch. If the bandwidth (i.e., the distance between the upper and lower frequency bounds) remains less than 1kHz, the sound is perceived as a noise band. However, if the bandwidth grows even larger, the frequencies of the individual partials become more discernable, creating an effect that one listener described as "metallic." When the bandwidth becomes very small (e.g. less than 10Hz), a whistle-like tone with a more discernable pitch emerges.

A custom OSW transform called AntonyVoice was created to implement the ANTONY-VOICE-MIGRATION algorithm. The transform has inlets for the upper and lower frequency bounds, and the migration rate, and outputs a sinusoidal model. The sinusoidal-model out-

Figure 6.7: OSW patch for *Antony*. The AntonyVoice transforms output sinusoidal models that change according to the migration algorithm. Wires connect from the three AntonyVoice transforms to the Multitrigger in order to signal that the migrations are complete. The Multitrigger connects back to the AntonyVoice transforms through the FanOut to restart the migration algorithms.

---

puts are connected to AddByIFFT transforms to synthesize waveform representations using TDAS, as illustrated in figure 6.7. The Gain transforms allow the performer to control the loudness of each waveform. The waveforms are mixed and sent to the audio output device. The three instances of AntonyVoice are linked together via the Multitrigger transform to synchronize the transition between epochs. When each algorithm completes its migration, it outputs an event to the Multitrigger. When the Multitrigger has received this event from all three AntonyVoice transforms, it sends a signal back to the transforms to begin the next

Figure 6.8: CPU usage in *Antony*. The average CPU time actually increases when the general reduction strategy is applied.

---

epoch and restart the migration process. The frequency-bound and rate parameters were updated remotely from a MIDI slider box (the MIDI input to each AntonyVoice transform is not shown in figure 6.7 to avoid unnecessary clutter). The Gain transforms are also controlled remotely via MIDI.

Antony is performed by changing the frequency bounds and migration rate in each voice in real time using the MIDI slider box. Several performances were created, with the MIDI events recorded to a sequencer during the performance so the same performance could be played back multiple times under different computational conditions. One of the performances was selected for the computational analyses and listener evaluations.

As illustrated in figure 6.8 the CPU usage actually *increased* when the generic re-

duction strategy used in the previous examples was applied. Because there is little variation in amplitude among partials, particularly if the bandwidth is less than 1kHz, approximately three quarters of partials are required to maintain three quarters of the total amplitude in the model. The computation saved by pruning only one quarter of the partials was not enough to offset the cost of the reduction algorithm.

Except for large anomalies at the beginning and end of the streams, the CPU times were more stable in this example than in the previous examples. The startup anomaly observed in both streams prior to 1.7 seconds is caused by artificial death and birth events in the model generated by the AntonyVoice transform. All partials above 100 had an initial amplitude of zero. They are interpreted by the synthesis transform as dead partials and not synthesized. As their frequencies and amplitudes were reset by the migration algorithm, the non-zero amplitudes were interpreted as rebirths. The gradual addition of these partials increased computation in the synthesis transforms until all partials had non-zero amplitudes. The sudden increase in computation after 42 seconds is due to an attempt by the sequencer to read beyond the end of recorded performance sequence, which ends at 42 seconds. Neither anomaly had a significant effect on the average or median CPU time over the duration of the piece. Both anomalies have since been corrected in the implementation.

A custom strategy is required to reduce computation in this example. However, because the models are generated algorithmically, there is additional knowledge available. In particular, it is known that large number of partials are randomly distributed in a fixed frequency range, and the models have a high spectral density. Human perception of such models was studied by Hartmann et al. [56]. They found that the spectrum became saturated after about 60 partials were added, and listeners could not discriminate between sound examples that contained more partials. They also found that the ability to discriminate between sounds with different numbers of partials increased as a function of bandwidth. Fewer partials are necessary when the bandwidth becomes small. Using these results, a custom reduction strategy was developed in which the number of partials in the

Figure 6.9: A reducible OSW patch for *Antony*. AntonyVoice has been replaced with ReducibleAntonyVoice, which integrates a custom reduction strategy into the migration algorithm.

---

model, N, is a function of bandwidth:

$$N = 60 + 4\log_2{}^2(\frac{f_{hi} - f_{lo}}{50}) \tag{6.1}$$

Using this formula, 60 is the minimum number of partials allowed for a model and the size increases in proportion to the square of the logarithm of the bandwidth. The squared-logarithm function was used to fit the model used by Hartmann et al., and the scaling factor of four was determined by informal listener evaluation. A complication of dynamically changing N is the change in loudness when partials are added or removed. This problem is

Figure 6.10: CPU usage in *Antony*, when the custom reduction strategy is applied. The average CPU time decreases by approximately 20% and 50% in the partially and fully reduced streams, respectively.

---

reduced by normalizing the amplitudes. The amplitude of each partial is scaled by N_{200}/N where N_{200} is the original number of partials and N is the reduced number of partials.

Figure 6.9 illustrates a reducible version of the patch used in *Antony*. The AntonyVoice transforms have been replaced with ReducibleAntonyVoice. When the perceptual scheduler determines that a reduction in computation is necessary, ReducibleAntonyVoice dynamically adjusts the size of the model used in the migration algorithm using equation 6.1 and the most recent upper and lower frequency bounds. As illustrated in figure 6.10, the average CPU time drops by approximately 25% and 50% for the partially reduced and maximally reduced streams, respectively. The startup anomaly before 1.7 seconds is still present.

Figure 6.11: Quality versus average CPU time in *Antony*. Listeners gave similar scores to both reduced-computation performances.

However, it is less pronounced in the reduced streams because the model sizes are reduced and fewer partials are reborn after the anomaly. The increase in CPU usage by the sequencer at the end of the piece is apparent in all three streams. Because the sequencer is not a reducible transform, the increase is approximately the same in all cases. However, in the partially reduced stream, the increase reached the upper bound on CPU time of 13μs per sample at 43.9 seconds. The perceptual scheduler notified the reducible transforms to reduce computation below the upper bound, resulting the observed decrease in CPU time.

The results of the listener evaluations are illustrated in figure 6.11. Both the fully reduced and partially reduced samples, with average CPU times of 11.3μs/sample and 7.3μs/sample, respectively, received similar average scores, suggesting that listeners noticed

some degradation when reductions were applied but increasing the level of reduction did not introduce additional strong artifacts. Listeners reported that a bell-like tone emerged very gradually from a noise band (i.e., as the bandwidth increased) in the original sample during the period between 16 and 19 seconds. However, in both of the reduced versions the bell-like tone entered more abruptly. During this period, the bandwidth of the voice increased quickly, so the size of the reduced model had to increase rapidly as well. The new partials are randomly assigned frequencies in the current band (i.e., as if the frequency had been updated by the migration algorithm). The amplitudes of the new partials are scaled using the *ampscale* function. This processes prevents artifacts from the normalization, which were more noticeable in early testing of the reduction algorithm. However, if the bandwidth increases too quickly between epochs, a large number of new partials will be initialized by this process and the migration of partials to the new frequency band will be perceived to occur more quickly.

Most listeners considered this a just-noticeable difference and gave both reduced-computation samples a score of four or more. However, three users deemed this a serious artifact and gave the samples a score of two, which lowered the mean scores of the partially reduced and fully reduced samples slightly below 4 to 3.85 and 3.77, respectively. Although these lower scores are considered statistical outliers (i.e., where outliers are defined as any elements whose distance above the third quartile or below the first quartile is more than 1.5 times the inter-quartile range), they are nonetheless reported in the results. We conclude that the custom strategy is successful in maintaining quality under reduced-computational conditions, with the caveat that normalization must be carefully balanced against the intended rate of timbral change in the migration algorithm.

Figure 6.12: Chords from *Constellation* used to play marimba models.

6.5 Example 4: Excerpts from *Constellation*, Ronald Bruce Smith

Constellation for orchestra and live electronics was composed by Ronald Bruce Smith and premiered by the Berkeley Symphony Orchestra in November of 2000 [114]. Smith was assisted by Timothy Madden, a visiting programmer at CNMAT, who created several Max/MSP patches for the performance [75]. In the piece, the orchestra is joined by a live electronics performer who uses a MIDI keyboard to control several real-time synthesis patches. Among the techniques used for the real-time synthesis was resonance modeling.

The two examples in this dissertation that studied the use of multiple resonance models in perceptual scheduling were based on musical material and synthesis processes developed for the premier of *Constellation*. The following subsections describe each of the examples and evaluate the effect of reductions on both computation and perceived audio quality.

6.5.1 Chords on a modified marimba model

In the first example, the keyboardist plays several large chords, as illustrated in figure 6.12, that are synthesized using six concurrent resonance models. The models are based on a marimba model that were was scaled using a brightness-model developed by Madden, to produce a more inharmonic sound. By playing six versions of the model simultaneously on different pitches, a single, harmonically-rich timbre is produced. The chords are perceived not as collections of pitches, but as spectral variations in the combined timbre.

Each of the six voices is implemented using a subpatch that contains an instance of the modified marimba model, as illustrated in figure 6.13. When a new MIDI note is received via the pitch inlet, it is converted to a frequency representation that is used to scale the model to the requested pitch. The JitterRes transform is used to randomly scale the initial amplitudes of the partials in the model between 70% and 130% of the original. (JitterRes is a port to OSW of Madden's `jerkres` object for Max/MSP.) The random scaling of each partial adds a sense of realism to the synthesized sound, because real instruments resonate with slightly different characteristics on successive excitations. The ReduceResonances transform applies the reduction algorithm to the scaled and amplitude-jittered model when instructed to do so by the scheduler. The reduced model is then sent to the Resonators transform to be synthesized using a resonant filterbank. Each incoming MIDI note also triggers an impulse that excites the resonant filterbank, resulting in a synthesized waveform of the model at the requested pitch. The waveform output from each of the six subpatches was mixed into a single waveform representing the entire chord.

The CPU usage of full-computation and two reduced-computation performances of the chords is illustrated in figure 6.14. Unlike the the sinusoidal-model examples, in which the number of partials changes, the number of resonance partials remains constant, so there is little change in CPU usage during the performances. The full-computation stream has an average CPU time of 19.3μs per sample and the fully reduced stream has an average

Figure 6.13: Subpatch used for synthesis of the modified marimba resonance model. Each MIDI note received via the pitch inlet simultaneously scales the model used by the resonant filterbank and triggers an impulse excitation. Six instances of the subpatch were used in parallel.

CPU time 6.7 μs per sample, so the average computation saved is approximately 12.6μs per sample, or 65%. The small spikes present in both streams at 0.40s, 0.65, 2.90s, 3.85s and 4.95 seconds correspond to the onset of each chord, which requires additional computation to change the parameters in the filterbank. The partially reduced performance, which was required to maintain computation at a level below two thirds of the full version (i.e., approximately 12μs per sample), had different average CPU times before and after the

Figure 6.14: CPU usage for full and reduced versions of marimba-model chords in *Constellation*.

---

large spike at 2.9 seconds. At 2.9 seconds, the filterbanks received modified parameters for the new chord. Because the CPU time remained consistently below the upper bound, the ReduceResonances transforms were allowed to gradually grow the size of the resonance models, as described in section 5.2. By the chord onset at 2.9 seconds, the size of the reduced resonance models allowed by the scheduler had grown significantly since the last chord onset. This increase caused the ReduceResonances transforms to pass larger models to the filterbanks, which led to the observed spike in CPU time. The scheduler detected this increase and chose a new reducible subset for generating reduced-size models on the next epoch. Although the second subset resulted in less savings, it still satisfied the constraint that computation be reduced below 12μs (i.e., two thirds of the original).

Figure 6.15: Quality versus average CPU time for the marimba-model chords in *Constellation*.

The results of the listener evaluations are illustrated in figure 6.15. The average CPU times of full-computation, partially reduced and fully reduced performances were 19.2μs/sample 10.6μs/sample and 6.7μs/sample, respectively. Most users perceived little degradation in the partially reduced performance. However, listeners were split on their perception of the fully reduced performance. Some listeners noticed little degradation in this version, while others noticed a decrease in the duration of the sustain of the tones because some partials with low amplitudes but long sustains were removed. Additionally, because the chords were perceived as single inharmonic tones, some listeners noticed a change in perceived pitch between the different samples. As described in the results for the tam-tam resonance model section 4.2.2, the pitches of such inharmonic tones are often

weak or ambiguous, and resolved differently if partials are removed from the spectrum [117]. Because this example showed little degradation in perceived quality when the CPU usage was reduced to almost half the original, we consider this level of reduction successful. However, it is recommended that more partials be retained in the reduced models by raising the contribution fraction in the reduction algorithm as described in section 4.3.

6.5.2 Glockenspiel and vibraphone models

In the second example based on *Constellation*, the keyboardist performs two monophonic lines, one with each hand. The right hand controls a glockenspiel model and the left hand controls a vibraphone model. Unlike the previous example, in which six instances of the same model were played as chords, two different models are used and the notes for the two models usually occur at different times. Similar to the previous example, notes from a MIDI sequence were routed according to MIDI channel to two subpatches that synthesized the notes using pitch-scaled versions of the glockenspiel and vibraphone models.

The CPU usage of full and two reduced-computation performances is illustrated in figure 6.16. The glockenspiel and vibraphone models contained 103 partials and 170 partials, respectively. Because the combined size of the two models in this example was almost as large as the six instances of the marimba used in the previous example (i.e., six times 48 partials, or 288 total), the CPU time required without reductions was almost as large. However, the fully reduced version required only one quarter the CPU time of the original version, which is a greater savings than observed for the previous example, and the greatest computational savings observed for any of the examples. The resonance-model reduction algorithm was able to reduce the sizes of the glockenspiel and vibraphone models to 14 and 3 partials, respectively, or 17 partials total. Although this number is slightly larger than the 12 partials used in the previous example when fully reduced, there was less overhead in this example because there were fewer synthesis and frequency- and amplitude-scaling operations, and also fewer reducible transforms for the perceptual scheduling algorithm

Figure 6.16: CPU usage for full and reduced synthesis of vibraphone and glockenspiel models.

---

to search. The partially reduced performance begins with maximum reductions on both models, but the CPU time increases on each note onset until it reaches the upper bound of approximately 7μs per sample. At 2.1 seconds, the CPU time exceeded the upper bound so the scheduler temporarily reduced the model sizes before allowing them to gradually grow again.

As illustrated in figure 6.17, listeners reported little or no decrease as quality as the average CPU time was partially reduced from 15.6μs/sample to 6.9μs/sample and then fully reduced to 4.2μs/sample. Because most of the energy in the two models was concentrated in only a few partials (i.e., similar to the marimba model in chapter 4 but unlike the modified marimba model in the previous section), most partials could be removed without

Figure 6.17: Quality versus average CPU time for performance of glockenspiel and vibraphone models in *Constellation*. This example received the highest listener scores as a function of reduced CPU time.

---

significant degradation. Additionally, because these models were more harmonic than the marimba model, the perceived pitch remained stable when partials were pruned. Because more computation was saved and less quality was lost than in any of the previous examples, this example was considered the most successful.

6.6 Discussion

Perceptual scheduling maintains strong upper bounds on execution times and real-time QoS guarantees in larger musical examples with changing computational requirements. Perceived audio quality depends on finding the right reduction strategies for the models be-

ing used. The generic sinusoidal-model reduction strategy developed in section 4.3 works well on single-tone models but causes more severe degradation in models containing multiple events or broader spectra. Solutions to this problem include parameterizing the reduction algorithm and developing additional reduction algorithms for different classes of models. However, these solutions require more expertise from composers and musical applications developers, or the development of machine-learning techniques that select the correct reduction strategies for different models.

Customized strategies are a viable option for algorithmically generated models where the mathematical and perceptual properties of the model are known in advance. A strategy for a sinusoidal model that was specific to a single musical work was used successfully to increase the computational savings and decrease degradation in quality.

Greater success at minimizing degradation in quality was observed for resonance models. Like algorithmically generated models, resonance models can be seen as special cases of sinusoidal models whose properties are known in advance. In particular, resonance models have a constant size (i.e., no birth or death events), fixed frequencies and exponentially-decaying amplitudes. These properties remain invariant under time-scaling operations like those used in section 6.3. The resonance-model reduction strategy can therefore prune partials using knowledge about the entire duration of the model as determined by the initial amplitude and decay rate. Although users can change the properties of resonance models in real time, they usually do so at a rate much less than the approximately 100Hz to 250Hz frame rate in analyzed sinusoidal models.

Chapter 7

Conclusions and Future Work

This chapter summarizes the work described in this dissertation. Section 7.1 reviews the motivations behind the development of perceptual scheduling. Section 7.2 describes the research contributions made by this thesis work. Section 7.3 proposes areas of future research, and section 7.4 summarizes the dissertation.

7.1 Review of Motivations and Design

Musicians who work with computers are increasingly turning from hardware towards software implementations on general purpose computers. Sound synthesis applications in software provide musicians richer and more complex control of sound in their performances and compositions. However, high computational requirements and the lack of QoS guarantees on general-purpose operating systems has frustrated many potential users of these applications. We introduced perceptual scheduling to meet this need.

In the perceptual scheduling framework, QoS guarantees are maintained by dynamically detecting potential failures and reducing computation in sound synthesis algorithms. A search problem is used to find a set of reducible synthesis algorithms whose combined computation savings lowers the CPU time below the upper bound for maintaining QoS guarantees (e.g., CPU time must be less than 20μs per sample to synthesize 44.1kHz audio

samples in real time). The system adapts to changing computational constraints caused by model changes, exceptions and increased CPU use by other processes. The perceptual scheduler dynamically increases computation in reducible algorithms when additional bandwidth becomes available, and decreases computation when the bandwidth becomes scarce.

Perceptually scheduled synthesis algorithms use measures of perceptual salience to judiciously reduce computational complexity while maintaining audio quality. This approach mirrors the use of perceptual encoders to maintain audio quality under reduced data bandwidth. Reduction strategies were developed for additive-synthesis and resonance-modeling algorithms based on the results of listener evaluations of models reduced using different measures of perceptual salience. Additive synthesis and resonance modeling were used because a clear relationship could be established between reduced model size, computational savings and degradation in audio quality.

7.2 Research Contributions

This section reviews the principal research contributions made by this thesis work. The major contributions include:

1. *A new language for developing real-time music and audio applications was introduced.*

The Open Sound World (OSW) real-time music programming environment was developed by the author to meet the needs of both performing musicans and computer-music researchers. OSW is a "visual dataflow programming language" in which users connect components called *transforms* into dataflow networks called *patches*. It includes a large set of standard transforms for basic event and signal processing, including transforms that manipulate sinusoidal-model and resonance-model representations. Applications can be developed incrementally by dynamically modifying patches even while they are running. In addition, the dynamic profiling and open architecture of the run-time system made OSW an ideal platform for the development and evaluation of perceptual scheduling. We intend

to release OSW for both musical and research use. More information can be obtained at `http://www.cnmat.berkeley.edu/OSW`.

2. *Perceptual scheduling was introduced as a novel approach to maintaining QoS guarantees in real-time music and audio applications. A perceptual scheduling framework dynamically detects failures and reduces computation.*

A perceptual scheduling framework was described as a general search problem in which a set of potential computational reductions is selected to lower CPU time below an upper bound. Best-fit and first-fit options are discussed, and the first-fit option is chosen to minimize the framework overhead. Different computation-reduction strategies for different synthesis algorithms and sound models can be used within this framework. The framework and an extensible set of reduction strategies was implemented in OSW.

3. *Model-reduction strategies were developed for sinusoidal and resonance models using the results of listening experiments.*

Strategies were developed for pruning partials from sinusoidal and resonance models based on measures of perceptual salience. Measures based on amplitude and masking effects were compared in separate controlled listening experiments for sinusoidal and resonance models. The results of both experiments indicated that there was little perceptual difference between reductions based on amplitude and masking effects. Because the strategies based on amplitude were more efficient, they were used to develop reduction algorithms to be used within the perceptual scheduling framework.

4. *The computational bandwidth saved by the perceptual scheduling framework and model-reduction strategies was measured. The CPU savings ranged from 33% to 75%.*

The CPU time used by additive synthesis and resonance modeling algorithms implemented in OSW was measured on a reference Intel Pentium II processor. CPU times measured with

and without perceptual scheduling enabled were compared for several additive synthesis and resonance models. The results indicated that significant computation was saved using perceptual scheduling. Computational savings was also measured for musical examples in which multiple models were synthesized simultaneously. The average CPU time saved while maintaining high audio quality (i.e., as reported by listener evaluations) ranged from 33% in an example with five simultaneous harpsichord sinusoidal models to 75% in an example using simultaneous glockenspiel and vibraphone models.

5. *The perceived quality of short perceptually scheduled musical examples was measured in a controlled listening experiment.*

Listeners were asked to compare the audio quality of five musical examples synthesized with and without perceptual scheduling enabled. Additionally, four of the examples were synthesized with the upper bound on execution time at two different levels. Overall, the results were encouraging, especially for the resonance-model examples which most listeners rated as good when the computation was reduced by as much as two thirds or three quarters. While quality was maintained in an example using single-tone sinusoidal models, an example using longer vocal phrases was significantly degraded when perceptual scheduling was enabled. The conclusion from this example was that additional sinusoidal-model reduction strategies should be developed for different classes of models, such as human voice. A custom reduction strategy developed for algorithmically generated models in the piece *Antony* also maintained quality under reduced computational bandwidth.

The overall lesson learned from developing the perceptual scheduling framework is that QoS failures can be averted dynamically and gracefully by targeted reductions in the computation used by synthesis algorithms. However, care must be taken when applying the reduction strategies to different classes of models. In general, the most computation can be saved without compromising quality when additional knowledge about the sound models is available. Such models include those that are algorithmically generated. Resonance models, which are a subset of sinusoidal models with predictable exponential decay and

fixed frequencies, can also be viewed as a special case where additional knowledge enhanced the development of a good reduction strategy.

7.3 Future Research Directions

This section describes possible future research directions based on the observed successes and shortcomings of the perceptual scheduling framework.

As observed in section 6.3 and discussed in the previous section, further research is required to develop additional reduction strategies for different classes of sinusoidal models. In particular, sinusoidal models of speech have been studied by others for use in compression systems because the spectral bandwidth of speech signals is narrower than general music signals and the rate of temporal and spectral change is limited [8]. Such an approach can be used to improve the quality of perceptual scheduling in examples like the Tibetan-song model described in section 6.3.

However, providing several reduction strategies requires users to choose among them explicitly or tweak parameters of a more general algorithm for a particular class of models. Such user expertise violates one of the main goals in this research of providing musicians an automatic system for dynamic reduction of computational bandwidth. A possible avenue of research involves the use of a machine-learning technique, such as neural networks [81] or graphical models [62], to automatically select an appropriate reduction strategy for a given sinusoidal model.

The algorithms and experiments described in the dissertation were designed to evaluate computational reductions that maintained high audio quality (i.e., little perceptible difference from the original). If the audio-quality requirements are relaxed, additional computational bandwidth can be saved using more aggressive reduction strategies. The development of such strategies and their evaluation in applications requiring less audio quality is another opportunity for future research.

Although the research described in this dissertation only applied perceptual schedul-

ing to the synthesis of sinusoidal and resonance models, it can be applied to other algorithms as well. A separate reduction strategy can be developed and evaluated for resonance models with complex non-impulse excitations. Such a strategy might require changes to the excitation signal in addition to pruning of resonance partials. Granular synthesis, which is also based on small divisions that have incremental effects on sound quality, can be made reducible under perceptual scheduling. A similar approach using listening experiments to develop a reduction algorithm is recommended. Perceptual scheduling can also be applied to pitch detection, in which accuracy of the estimated pitch can be traded for reduced computational bandwidth in some applications (e.g., if all detected pitches are quantized to the traditional Western twelve-tone resolution), and video processing, in which image quality is traded for computational complexity.

7.4 Summary

This dissertation described the design, implementation and measurement of the perceptual scheduling framework for real-time music and audio applications. Perceptual scheduling was motivated by the need to reduce computation in applications that are too computationally expensive and systems where reliable real-time performance cannot be guaranteed when all available computational bandwidth is being used. Additive synthesis and resonance modeling were selected as the target applications for this research because the relationship between model size, computational complexity and perceived quality can be quantified. Perceptual scheduling was implemented in the OSW real-time music language. Computational performance was measured for additive-synthesis and resonance-modeling algorithms in OSW with and without perceptual scheduling enabled. Comparisons of these performance measurements indicated that the perceptual scheduling framework successfully bounded CPU times in these applications. The perceived quality of the short musical examples synthesized with and without perceptual scheduling was compared by listeners in controlled experiments. Degradation in quality was measured as a function of CPU time

saved. The results of these experiments indicate that perceptual scheduling combined with suitable reduction strategies can save computation and avoid potential real-time QoS failures while maintaining audio quality. Future research will develop additional reduction strategies for different classes of sinsusoidal models. Other researchers are encouraged to extend the use of perceptual scheduling to other computer-music applications such as granular synthesis and pitch detection, as well as other perceptually motivated applications such as video processing.

Bibliography

[1] ISO 226: Acoustics - Normal Equal-loudness Contours, 1987.

[2] ITU-T P.800: Methods for Subjective Determination of Transmission Quality, 1993.

[3] Inquiry Board Traces Ariane 5 Failure to Overflow Error, 1996. Society for Industrial and Applied Mathematics `http://www.siam.org/siamnews/general/ariane.htm`.

[4] The Complete MIDI 1.0 Detailed Specification. Protocol specification, MIDI Manufacturers Association, 1996.

[5] The Relationship of Dynamic Range to Data Word Size in Digital Audio Processing. Technical report, Analog Devices, Inc., 1998. `http://www.analog.com/publications/whitepapers/products/32bit_wa.html`.

[6] Honda Insight Engineering, 2001. `http://www.honda2001.com/models/insight/engineering.html`.

[7] AES. AES Recommended Practice for Digital Audio Engineering - Serial Transmission Format for Linearly Represented Digital Audio Data. *Journal of the Audio Engineering Society*, 33(12):975–84, 1985. English.

[8] S. Ahmadi and A. S. Spanias. New techniques for sinusoidal coding of speech at 2400 bps. In A. Singh, editor, *Conference Record of The Thirtieth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 770–4, Pacific Grove, CA, USA, 1996. IEEE Comput. Soc. Press.

[9] J. Allen. Computer Architecture for Digital Signal Processing. *Proceedings of the IEEE*, 73(5):852–73, 1985. Article.

[10] D. P. Anderson and R. Kuivila. Acurately Timed Generation of Discrete Musical Events. *Computer Music Journal*, 10(3):48–56, 1986.

[11] R. Avizienis, A. Freed, T. Suzuki, and D. Wessel. Scalable Connectivity Processor for Computer Music Performance Systems. In *International Computer Music Conference*, Berlin, Germany, 2000.

[12] J. S. Bach. Prelude and Fugue No. 22 in Bb minor. In *Well-Tempered Clavier, Book I.* 1722. Urtext edition, G. Henle Verlag, Munich Germany, 1978.

[13] J. W. Beauchamp. Unix Workstation Software for Analysis, Graphics, Modification, and Synthesis of Musical Sounds. In *Audio Engineering Society Convention*, Berlin, Germany, 1993.

[14] E. Brandt and R. Dannenberg. Low-Latency Music Software Using Off-the-Shelf Operating Systems. In *International Computer Music Conference*, pages 137–140, Ann Arbor, MI, 1998.

[15] E. Brandt and R. B. Dannenberg. Time in Distributed Real-Time Systems. In *International Computer Music Conference*, Beijing, China, 1999.

[16] J. Brown. Give It Up or Turnit A Loose, 1970. On *James Brown Funk Power, 1970: A Brand New Thang*, Polygram Records, Inc., 1996.

[17] J. Bruno, E. Gabber, B. Özden, and A. Silberschatz. Move-To-Rear List Scheduling: A New Scheduling Algorithm for Providing QoS Guarantees. In *ACM Multimedia 97*, Seattle, WA, 1997.

[18] R. Caussé, P. Dérogis, and O. Warusfel. Radiation of Musical Instruments and Improvement of the Sound Diffusion Techniques for Synthesized, Recorded or Amplified

Sounds (Revisited). In *International Computer Music Conference*, Banff, Canada, 1995.

[19] A. Chaudhary. Band-limited Simulation of Analog Synthesizer Modules by Additive Synthesis. In *105th AES Convention*, San Francisco, CA, 1998.

[20] A. Chaudhary. *OpenSoundEdit: An Interactive Visualization and Editing Framework for Timbral Resources.* Masters thesis, Electrical Engineering and Computer Science, University of California, Berkeley, CA, 1998. `http://www.cnmat.berkeley.edu/ amar/MSThesis`.

[21] A. Chaudhary, A. Freed, S. Khoury, and D. Wessel. A 3D Graphical User Interface for Resonance Modeling. In *International Computer Music Conference*, Ann Arbor, Michigan, 1998.

[22] A. Chaudhary, A. Freed, and D. Wessel. Exploiting Parallelism in Real-Time Music and Audio Applications. In *International Symposium on Computing in Object-Oriented Parallel Environments*, San Francisco, CA, 1999.

[23] A. Chaudhary, A. Freed, and M. Wright. An Open Architecture for Real-time Audio Processing Software. In *107th AES Convention*, New York, 1999. `http://www.cnmat.berkeley.edu/OSW`.

[24] Chowning. Stria, 1977.

[25] J. M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. In Curtis Roads and John Strawn, editors, *Foundations of Computer Music*, pages 6–29. MIT Press, Cambridge, MA, 1985.

[26] E. Clarke. Rhythm and Timing in Music. In Diana Deutsch, editor, *The Psychology of Music*, pages 473–500. Academic Press, San Diego, 1999.

[27] E. F. Clarke. The Perception of Expressive Timing in Music. *Psychological Research*, 51:2–9, 1989.

[28] D. J. Collinge. MOXIE: A Language for Computer Music Performance. In *International Computer Music Conference*, pages 217–220, 1984.

[29] P. Cook. Synthesis Toolkit in C++. In *SIGGRAPH*, New York, NY, 1996. ACM.

[30] P. R. Cook. Physically Inspired Sonic Modeling (PhISM): Synthesis of Percussive Sounds. *Computer Music Journal*, 21:38–49, 1997.

[31] M. S. Corrington. Variation of Bandwidth with Modulation Index in Frequency Modulation. In Klapper, editor, *Papers on Frequency Modulation*. Dover, 1970.

[32] R. Dannenberg. Real-Time Scheduling and Computer Accompaniment. In Max Matthews and John Pierce, editors, *Current Research in Computer Music*. MIT Press, Cambridge, MA, 1989.

[33] R. Dannenberg and E. Brandt. A Flexible Real-Time Software Synthesis System. In *International Computer Music Conference*, Hong Kong, 1996.

[34] R. B. Dannenberg. Machine tongues XIX: Nyquist, a language for composition and sound synthesis. *CMJ*, 21(3):50–60, 1997.

[35] R. H. Davis. *Synthesis of steady-state signal components by an all-digital system*. Doctoral dissertation, Maryland, 1974.

[36] F. Déchelle et al. jMax: A New JAVA-Based Editing and Control System for Real-time Musical applications. In *International Computer Music Conference*, Ann Arbor, MI, 1998.

[37] G. DiGiugno. A 256 Digital Oscillator Bank. In *Computer Music Conference*, Cambridge, Massachusetts: M.I.T., 1976.

[38] M. Dolson. The Phase Vocoder: A Tutorial. *Computer Music Journal*, 10(4):14–17, 1986.

[39] R. Dudas. NVM: A Modular Real-time Physical Modelling Synthesis System for MSP. In *ICMC*, Ann Arbor, MI, 1998.

[40] D. Durham and R. Yavatkar. *Inside the Internet's Resource reSerVation Protocol*. Wiley Computer Publishing, New York, 1999.

[41] K. Fitz, 2001. Personal Communication.

[42] K. Fitz, L. Haken, and P. Christensen. Transient Preservation under Transformation in an Additive Sound Model. In *International Computer Music Conference*, Berlin, Germany, 2000.

[43] A. Freed. Codevelopment of User Interface, Control and Digital Signal Processing with the HTM Environment. In *5th International Conference on Signal Processing Applications and Technology*, pages 1179–83 vol.2, Dallas, TX, USA, 1994. DSP Associates. 18-21 Oct. 1994.

[44] A. Freed. Real-Time Inverse Transform Additive Synthesis for Additive and Pitch Synchronous Noise and Sound Spatialization. In *AES 104th Convention*, San Francisco, CA, 1998. AES.

[45] A. Freed. Musical Applications of Resonance Models, 1999. `http://cnmat.cnmat.berkeley.edu/Research/Resonances/`.

[46] A. Freed and A. Chaudhary. Music Programming with the new Features of Standard C++. In *International Computer Music Conference*, pages 244–247, Ann Arbor, MI, 1998. `http://www.cnmat.berkeley.edu/Research`.

[47] A. Freed, X. Rodet, and P. Depalle. Synthesis and control of hundreds of sinusoidal partials on a desktop computer without custom hardware. In *Fourth International*

Conference on Signal Processing Applications and Technology ICSPAT '93, pages 1024–30 vol.2, Santa Clara, CA, USA, 1993. DSP Associates. 28 Sept.-1 Oct. 1993.

[48] A. Freed and D. Wessel. Communication of Musical Gesture using the AES/EBU Digital Audio Standard. In *International Computer Music Conference*, Ann Arbor, Michigan, 1998.

[49] A. Freed and M. Wright. CAST: CNMAT's Additive Synthesis Tools. Technical report, CNMAT, 1998. `http://www.cnmat.berkeley.edu/CAST`.

[50] D. Gabor. Acoustical quanta and the theory of hearing. *Nature*, 159:591–594, 1947.

[51] G. Garcia and J. Pampin. Data Compression of Sinusoidal Modeling Parameters Based on Psychoacoustic Masking. In *International Computer Music Conference*, Beijing, 1999.

[52] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queueing: A Scheduling Algorithm for Integrated Sevices Packet Switching Networks. In *SIGCOMM*, 1996.

[53] L. Haken. Computational Methods for Real-time Fourier Synthesis. *IEEE Transactions on Signal Processing*, 40(9):2327–2329, 1992.

[54] L. Haken, 2001. Personal Communication.

[55] L. Haken, K. Fitz, and P. Christensen. Beyond Traditional Sampling Synthesis: Real-time Timbre Morphing using Additive Synthesis. In James W. Beauchamp, editor, *Sound of Music: Analysis, Synthesis and Perception*. Springer-Verlag, 2000.

[56] W. M. Hartmann, S. McAdams, A. Gerzso, and P. Boulez. Discrimination of Spectral Density. *Journal of the Acoustical Society of America*, 79(6):1915–1925, 1986.

[57] V. Iyer. *Microstructures of Feel, Macrostructures of Sound: Embodied Cognition in West African and African-American Musics*. Doctoral disserta-

tion, Technology and the Arts, University of California, Berkeley, CA, 1998. http://cnmat.cnmat.berkeley.edu/People/Vijay/%20THESIS.html.

[58] V. Iyer, J. Bilmes, D. Wessel, and M. Wright. A Novel Representation for Rhythmic Structure. In *International Computer Music Conference*, pages 97–100, Thessaloniki, Hellas, 1997.

[59] D. Jaffe. Ensemble Timing in Computer Music. *Computer Music Journal*, 9(4):38–48, 1985.

[60] D. H. Jameson. Building Real-Time Music Tools Visually with Sonnet. In *1996 IEEE Real-Time Technology and Applications Symposium*, Boston, MA, 1996.

[61] N. Jayant, J. Johnston, and R. Safranek. Signal compression based on models of human perception. *Proc. IEEE*, 81(10):1385–1422, 1993.

[62] M. I. Jordan, editor. *Learning in Graphical Models*. MIT Press, Cambridge, MA, 1999.

[63] S. J. Kaplan. Developing a Commercial Digital Sound Synthesizer. In *The Music Machine*. MIT Press, Cambridge, MA, 1989.

[64] K. Karplus and A. Strong. Digital Synthesis of Plucked String and Drum Timbres. *Computer Music Journal*, 7(2), 1983.

[65] A. Kaup, A. Freed, S. Khoury, and D. Wessel. Volumetric Modeling of Acoustic Fields for Musical Sound Design in a New Sound Spatialization Theatre. In *International Computer Music Conference*, Beijing, China, 1999. ICMA.

[66] P. Lansky. The Architecture and Musical Logic of Cmix. In *International Computer Music Conference*, pages 91–94, Glasgow, 1990.

[67] J. Laroche. The use of the matrix pencil method for the spectrum analysis of musical signals. *Journal of the Acoustical Society of America*, 94(4):1958–65, 1993. Article.

[68] J. Laroche. Time and pitch scale modification of audio signals. In Mark Kahrs and Karlheinz Brandenburg, editors, *Applications of Signal Processing to Audio and Acoustics*, pages 279–310. Kluwer Academic, New York, 1998.

[69] J. Lazzaro and J. Wawryznek. MPEG-4 Structured Audio: Developer Tools, 2000. `http://www.cs.berkeley.edu/ lazzaro/sa`.

[70] E. A. Lee et al. Heterogeneous Concurrent Modeling and Design in Java. Technical Report UCB/ERL M98/72, EECS, University of California, November 23, 1998 1998. `http://ptolemy.eecs.berkeley.edu`.

[71] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, 1995.

[72] B. Li and K. Nahrstedt. A Control-based Middleware Framework for Quality of Service Adaptations. *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, 1999.

[73] R. A. Likert. A Technique for the Measurement of Attitudes. *Archives of Psychology*, (140):5–55, 1932.

[74] T. Madden and J. Beauchamp. Real Time and Non-Real Time Analysis of Musical Sounds on a Power Macintosh. Beijing, China, 1999.

[75] T. Madden, R. B. Smith, M. Wright, and D. Wessel. Preparation for Interactive Live Computer Performance in Collaboration with a Symphony Orchestra. Technical report, Center for New Music and Audio Technologies, University of California, 2001.

[76] M. A. Marks. Resource Allocation in an Additive Synthesis System for Audio Waveform Generation. In *International Computer Music Conference*, pages 378–382, San Francisco, CA, 1988.

[77] M. V. Matthews. *The Technology of Computer Music*. MIT Press, Cambridge, MA, 1969.

[78] R. J. McAulay and T. F. Quatieri. Speech analysis/synthesis based on a sinusoidal representation. *IEEE ASSP*, 34(4):744–754, 1986.

[79] J. McCartney. SuperCollider: A new real-time sound synthesis language. In *Int. Comp. Music Conf.*, pages 257–258, Hong Kong, 1996. International Computer Music Association.

[80] K. Meyer-Patel. *A Parallel Software-Only Video Effects Processing System*. Doctoral dissertation, Computer Science, University of California, Berkeley, CA, 1999. `http://bmrc.berkeley.edu/research/publications/1999/155/index.html`.

[81] W. T. Miller, R. S. Sutton, and P. J. Werbos. *Neural Networks for Control*. Bradford Books. The MIT Press, Cambridge, Mass, 1990.

[82] D. Mills. Network Time Protocol (Version 3) Specification, Implementation, and Analysis. Technical Report Internet RFC 1305, 1992.

[83] D. Mills. Simple Network Time Protocol (SNTP) Version 4 for Ipv4, Ipv6 and OSI. Technical Report Internet RFC 2030, 1996. `http://sunsite.auc.dk/RFC/rfc2030.html`.

[84] F. R. Moore. *Elements of Computer Music*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[85] J. A. Moorer. The use of the phase vocoder in computer music applications. *Journal of the Audio Engineering Society*, 26(1-2):42–5, 1978. Article.

[86] E. Moulines and F. Charpentier. Pitch-Synchronous Waveform Processing Techniques for Text-To-Speech Synthesis using Diphones. *Speech Communication*, 9:453–467, 1990.

[87] M. Nicholl. Good Vibrations. *Invention and Technology*, 1993.

[88] H. Park, S. Van Huffel, and L. Elden. Fast algorithms for exponential data modeling. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages IV/25–8 vol.4, Adelaide, SA, Australia, 1994.

[89] N. Porcaro, P. Scandalis, J. O. Smith, D. A. Jaffe, and T. Stilson. SynthBuilder – A Graphical Real-time Synthesis, Processing and Performance System. In *International Computer Music Conference*, Banff, Canada, 1995.

[90] Y. Potard, P.-F. Baisnée, and J.-B. Barrière. Experimenting with Models of Resonance Produced by a New Technique for the Analysis of Impulsive Sounds. In *International Computer Music Conference*, pages 269–274, La Haye, 1986.

[91] J. Princen and A. Bradley. Analysis/synthesis Filter Band Design Based on Time-domain Aliasing Cancellation. *IEEE Trans. Acoustics, Speech and Signal Processing*, 34:1151–1161, 1986.

[92] M. Puckette. The Patcher. In *Proceeings of the 14th International Computer Music Conference*, Koln, 1988.

[93] M. Puckette. Pure Data: Another Integrated Computer Music Environment. In *Second Intercollege Computer Music Concerts*, pages 37–41, Tachikawa, Japan, 1996.

[94] M. Puckette, T. Apel, and D. Zicarelli. Real-time audio analysis tools for Pd and MSP. In *International Computer Music Conference*, pages 109–112, Ann Arbor, MI, 1998.

[95] R. Puri and K. Ramchandran. Multiple Description Source Coding Through Forward Error Correction Codes. In *Proceedings of the 33rd Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, 1999.

[96] S. R. Quackenbush, T. P. I. Barnwell, and e. al. *Objective Measures of Speech Quality.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[97] T. F. Quatieri and R. J. McAulay. Shape invariant time-scale and pitch modification of speech. *IEEE Transactions on Signal Processing*, 40(3):497–510, 1992. Article Access restricted.

[98] B. H. Repp. Probing the Cognitive Representation of Musical Time: Structual Constraints on the Perception of Timing Perturbations. *Cognition*, 44:241–281, 1992.

[99] B. H. Repp. Musical Motion in Perception and Performance. In David A. Resonbaum and Charles E. Collyer, editors, *Timing of Behavor: Neural, Psychological and Computational Perspectives*, pages 125–144. MIT Press, Cambridge, MA, 1998.

[100] M. Reyoto. Fugue No. 22 in Bb minor (BWV 867), 2001. Standard MIDI File at Classical Music Archive, Pierre R. Schwob, Classical Archives, LLC. `http://www.prs.net/bach.html`.

[101] M. R. Riedel. *Kiss the Shattered Glass and Time, Space and the Synthesis of Illusion: the Creation of Space and Sonic Mass in 'Kiss the Shattered Glass'.* Doctoral dissertation, Rutgers University, 2001.

[102] J.-C. Risset and D. Wessel. Exploration of Timbre by Analysis Synthesis. In Diana Deutsch, editor, *The Psychology of Music.* Academic Press, San Diego, 1999.

[103] C. Roads. Granular Synthesis of Sound. In C urtis Roads and John Strawn, editors, *Foundations of Computer Music*, pages 145–159. MIT Press, Cambridge, MA, 1988.

[104] X. Rodet. Musical sound signal analysis/synthesis: sinusoidal-plus-residual and elementary waveform models. *Applied Signal Processing*, 4(3):131–41, 1997. Article Springer-Verlag.

[105] X. Rodet, Y. Potard, and J.-B. Barrière. The CHANT Project : From Synthesis of the Singing Voice To Synthesis in General. *Computer Music Journal*, 8(3):15–31, 1984.

[106] L. A. Rowe and B. C. Smith. A Continuous Media Player. In *3rd International Workshop on Network and Operating System Support for Digital Audio and Video*, San Diego, CA, 1992.

[107] C. Scaletti. The Kyma/Platypus Computer Music Workstation. *Computer Music Journal*, 15(3):41–49, 1989.

[108] E. D. Scheirer. Structured Audio, Kolmogorov Complexity, and Generalized Audio Coding (in press). *IEEE Transactions on Speech and Audio Processing*, 2000.

[109] E. D. Scheirer and B. L. Vercoe. SAOL: The MPEG-4 Structured Audio Orchestra Language. *Computer Music Journal*, 23(2), 1999.

[110] X. Serra and III J. Smith. Spectral modeling synthesis: a sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *Computer Music Journal*, 14(4):12–24, 1990.

[111] S. Shlien. Guide to MPEG-1 Audio Standard. *IEEE Trans. on Broadcasting*, 40(4):206–218, 1994.

[112] J. O. Smith. Physical Modeling Using Digital Waveguides. *Computer Music Journal*, 16(4), 1992.

[113] J. O. Smith and P. Gossett. A Flexible Sampling-Rate Conversion Method. In *IEEE ICASSP*, volume 2, pages 19.4.1–19.4.2, San Diego, CA, 1984.

[114] R. B. Smith. Constellation for orchestra and live electronics, 2000. Premiered in November 2000 by the Berkeley Symphony Orchestra.

[115] J.-F. Susini, L. Hazard, and F. Boussinot. Distributed Reactive Machines. In *Fifth International Conference on Real-Time Computing Systems and Applications*, pages 267–274, Hiroshima, Japan, 1998. IEEE.

[116] E. Terhardt. Psychoacoustic evaluation of musical sounds. *Perception and Psychophysics*, 23:483–492, 1978.

[117] E. Terhardt, G. Stoll, and M. Seewann. Pitch of Complex Signals According to Virtual-pitch Theory: Tests, Examples and Predictions. *Journal of the Acoustical Society of America*, 71:671–678, 1982.

[118] D. Topper. RTcmix for Linux. 1. *Linux Journal*, (78):166–172, 2000.

[119] K. Tsutsui et al. ATRAC: Adaptive Transform Acoustic Coding for MiniDisc. In *93rd Audio Engineering Society Convention*, San Francisco, CA, 1992. `http://www.minidisc.org/aes_atrac.html`.

[120] M. Tsuzaki and R. D. Patterson. Jitter Detection: A Brief Review and Some New Experiments. In A. Palmer, R. Summerfield, R. Meddis, and A. Rees, editors, *Proceedings of the Symposium on Hearing*, Grantham, UK, 1997.

[121] B. Vercoe and D. P. W. Ellis. Real-time CSound: Software Synthesis with Sensing and Control. In *International Computer Music Conference*, pages 209–211, Glasgow, 1990.

[122] B. Vercoe et al. The Csound Manual (version 3.47): A Manual for the Audio Processing System and Supporting Programs with Tutorials, 1997. `http://mitpress.mit.edu/e-books/csound/csoundmanual/TITLE.html`.

[123] S. Vernon. Design and Implementation of AC-3 Coders. *IEEE Tr. Consumer Electronics*, 41(3), 1995.

[124] H. von Helmholtz and A. J. Ellis. *On the sensations of tone as a physiological basis for the theory of music*. Dover Publications, New York,, 1875. 2d English ed., translated, thoroughly rev. and corrected, rendered conformal to the 4th (and last) German ed. of 1877, with numerous additional notes and a new additional appendix bringing down information to 1885, and especially adapted to the use of music students, by Alexander J. Ellis. With a new introd. (1954) by Henry Margenau.

[125] G. H. Wakefield. A Mathematical/Psychometric Framework for Comparing the Perceptual Response to Different Analysis-synthesis Techniques: Ground Rules For A Synthesis Bake-Off. In *International Computer Music Conference*, Berlin, 2000.

[126] J. C. Wawrzynek, T.-M. Lin, C. A. Mead, L. H., and D. L. A VLSI Approach to Sound Synthesis. In *International Computer Music Conference*, Paris, 1984.

[127] D. Wessel. Antony, 1977. On *Compter Music Currents 10*, Wego Schallplatten GmbH, Mainz, Germany, 1992.

[128] D. Wessel, C. Drame, and M. Wright. Removing the Time Axis from Spectral Model Analysis-Based Additive Synthesis: Neural Networks versus Memory-Based Machine Learning. Ann Arbor, Michigan, 1998. ICMA.

[129] D. Wessel, M. Wright, and S. A. Khan. Preparation for Improvised Performance in Collaboration with a Khyal Singer. In *International Computer Music Conference*, Ann Arbor, Michigan, 1998.

[130] D. L. Wessel, P. Lavoie, L. Boynton, and Y. Orlarey. MIDI-Lisp: A Lisp-based programming environment for MIDI on the Macintosh. In *AES 5th International Conference: Music and Digital Technology*, volume 1, pages 185–197, Los Angeles, 1987. Audio Engineering Society, New York.

[131] M. Wright. Implementation and Performance Issues with OpenSound Control. In *International Computer Music Conference*, Ann Arbor, Michigan, 1998.

[132] M. Wright, A. Chaudhary, A. Freed, S. Khoury, and D. Wessel. Audio Applications of the Sound Description Interchange Format Standard. In *107th AES Convention*, New York, 1999. `http://www.cnmat.berkeley.edu/SDIF`.

[133] M. Wright et al. Panel Session on Analysis/Synthesis Techniques. Berlin, 2000.

[134] M. Wright, S. Khoury, R. Wang, and D. Zicarelli. Supporting the Sound Description Interchange Format in the Max/MSP Environment. In *International Computer Music Conference*, Beijing, China, 1999.

[135] M. Wright and E. Scheirer. Cross-Coding SDIF into MPEG-4 Structured Audio. In *International Computer Music Conference*, Beijing, China, 1999.

[136] D. Zicarelli. An Extensible Real-Time Signal Processing Environment for Max. In *International Computer Music Conference*, pages 463–466, Ann Arbor, MI, 1998. `http://www.cycling74.com`.

[137] E. Zwicker and H. Fastl. *Psychacoustics: Facts and Models, 2nd Edition*, volume 22 of *Springer Series in Information Sciences*. Springer-Verlag, Berlin, 1999.

[138] E. Zwicker and T. Zwicher. Audio Engineering and Psychoacoustics: Matching Signals to the Final Receiver, the Human Auditory System. *J. Audio Eng. Soc*, 39(3):115–126, 1991.

Appendix A

Supplementary Audio CD

The tracks on the supplementary audio CD are listed below.

Sinusoidal models (Chapter 4)

1. Suling model, original (150 partials) [0:08]
2. Suling model, 113 partials [0:08]
3. Suling model, 75 partials [0:08]
4. Suling model, 56 partials [0:08]
5. Suling model, 38 partials [0:08]
6. Suling model, 19 partials [0:08]
7. Suling model, 9 partials [0:08]
8. Suling model, 5 partials [0:08]
9. Suling model, 2 partials [0:08]
10. Berimbao model, original (200 partials) [0:04]
11. Berimbao model, 150 partials [0:04]
12. Berimbao model, 100 partials [0:04]
13. Berimbao model, 75 partials [0:04]
14. Berimbao model, 50 partials [0:04]
15. Berimbao model, 25 partials [0:04]
16. Berimbao model, 13 partials [0:04]
17. Berimbao model, 6 partials [0:04]
18. Berimbao model, 3 partials [0:04]

19. James Brown recording, original sample [0:06]
20. James Brown model, original (240 partials) [0:06]
21. James Brown model, 180 partials [0:06]
22. James Brown model, 120 partials [0:06]
23. James Brown model, 90 partials [0:06]
24. James Brown model, 60 partials [0:06]
25. James Brown model, 30 partials [0:06]
26. James Brown model, 15 partials [0:06]
27. James Brown model, 8 partials [0:06]
28. James Brown model, 4 partials [0:06]

Resonance models (Chapter 4)

29. Marimba model, original (48 partials) [0:02]
30. Marimba model, 37 partials [0:02]
31. Marimba model, 31 partials [0:02]
32. Marimba model, 25 partials [0:02]
33. Marimba model, 19 partials [0:02]
34. Marimba model, 13 partials [0:02]
35. Marimba model, 10 partials [0:02]
36. Marimba model, 7 partials [0:02]
37. Marimba model, 4 partials [0:02]
38. Marimba model, 2 partials [0:02]
39. Bass model, original (59 partials) [0:04]
40. Bass model, 45 partials [0:04]
41. Bass model, 37 partials [0:04]
42. Bass model, 30 partials [0:04]
43. Bass model, 23 partials [0:04]
44. Bass model, 15 partials [0:04]
45. Bass model, 12 partials [0:04]
46. Bass model, 8 partials [0:04]
47. Bass model, 4 partials [0:04]
48. Bass model, 2 partials [0:04]
49. Tam-tam model, original (183 partials) [0:08]
50. Tam-tam model, 138 partials [0:08]
51. Tam-tam model, 115 partials [0:08]

52. Tam-tam model, 92 partials [0:08]
53. Tam-tam model, 69 partials [0:08]
54. Tam-tam model, 46 partials [0:08]
55. Tam-tam model, 35 partials [0:08]
56. Tam-tam model, 23 partials [0:08]
57. Tam-tam model, 12 partials [0:08]
58. Tam-tam model, 6 partials [0:08]

Additional models for performance measurements (Chapter 5)

59. Steve Coleman saxophone lick (sinusoidal model) [0:02]
60. Shafqat Ali Khan sinusoidal model [0:05]
61. "Angry cat" sinusoidal model [0:06]
62. Piano A0 resonance model [0:08]

Musical Examples (Chapter 6)

63. Excerpt from Fugue in B flat minor, J.S. Bach (BWV 867) [0:34]
64. Fugue, with computational reductions [0:34]
65. Time-scale Improvisation on Recording of Tibetan Singing (Tsering Wangmo) [0:28]
66. Tibetan Singing, with partial computational reductions [0:28]
67. Tibetan Singing, with full computational reductions [0:28]
68. *Antony*, David Wessel [0:45]
69. *Antony*, with partial computational reductions [0:45]
70. *Antony*, with full computational reductions [0:45]
71. Marimba-like chords from *Constellation*, Ronald Bruce Smith [0:07]
72. *Constellation* marimba, with partial computational reductions [0:07]
73. *Constellation* marimba, with full computational reductions [0:07]
74. Glockenspiel and vibraphone from *Constellation*, Ronald Bruce Smith [0:21]
75. Glockenspiel and vibraphone, with partial computational reductions [0:21]
76. Glockenspiel and vibraphone, with full computational reductions [0:21]