Recursive Oscillators on a Fixed-Point Vector Microprocessor for High Performance Phase-Accurate Real-Time Additive Synthesis

by

Todd David Hodes

Research Project

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science, Plan II

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

December 1997

Committee in charge:

Professor John Wawrzynek, Research Advisor
Professor Randy Katz

The report of Todd David Hodes is approved:

Chair | Date

Date

University of California at Berkeley

1997

Recursive Oscillators on a Fixed-Point Vector Microprocessor for High Performance Phase-Accurate Real-Time Additive Synthesis

Recursive Oscillators on a Fixed-Point Vector Microprocessor for High Performance Phase-Accurate Real-Time Additive Synthesis

by

Todd David Hodes

Abstract

The are many benefits in the use of *additive synthesis* for sound production in computer music applications. The challenge of the technique lies in its voracious appetite for separately controllable sinusoidal partials.

This paper summarizes our work developing a real-time additive synthesis engine supporting hundreds of simultaneous partials. It is the implementation for the T0 vector microprocessor. The goal was to provide significantly more real-time partials than were available using conventional general-purpose hardware architectures. The major features of T0 that drive the design is the vector ISA and the use of fixed-point arithmetic. The explicit parallelism of the vector ISA led us to use parallel recursive oscillators. The 16b fixed-point arithmetic required adapting the recursive oscillators to provide additional accuracy. The modified oscillator is a two-pole filter that maintains frequency precision at a cost of one additional operation per filter sample. The new filter's error properties are expressly imperfect, explicitly matched to use in the context of digital audio rather than general-purpose applications. We briefly describe the controlling synthesizer software, the control structure for feeding the oscillators, fast initialization (needed for short additive synthesis windows), and applicability to related architectures. We present algorithm performance analysis and measurements of the implementation, focusing on how chip features affected algorithm design choices. The technique achieves 608 simultaneous real-time partials (at 44.1KHz) with only a 40MHz clock performing 8 operations per cycle (peak), or about 1.5 cycles per partial per sample.

Acknowledgments

I would like to most heartily thank Professor John Wawrzynek for overseeing this research, providing most of the direction for it, and introducing me to the community of researchers at the Center for New Music and Audio Technologies. I would also like to thank John Hauser, Adrian Freed, and David Wessel for their extensive assistance and interest. Thanks to Randy Katz for allowing me to work on this after it dragged out longer than it ever should have.

Contents

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**
1.1 Additive Synthesis 1
1.2 System Framework 2

2 Method Description **4**
2.1 Interface and Framing 4
2.2 Oscillators 6
2.2.1 Recasting the Oscillator Form 6
2.2.2 Initialization 8
2.3 Error Analysis 10
2.3.1 Absolute Error 10
2.3.2 Relative error 11
2.3.3 Rounding error 12
2.3.4 Effects of framing 12
2.4 Example output 14

3 Performance Analysis **16**
3.1 Analytic 16
3.2 Measured 18
3.3 Other Metrics 20

4 Programming Considerations and Variations **22**
4.1 Vector register allocation 22
4.2 Amplitude Updates 23
4.3 I/O 23
4.4 Re-implementation performance 24
4.5 Application to related architectures 24
4.5.1 Other Vector Processors 24
4.5.2 “Multimedia” ISAs 24

5 Conclusions **26**

A Annotated Inner Stripmine Loop **27**

Bibliography **29**

List of Figures

1.1 An example analysis/synthesis framework including real-time procedures for live performance 3

2.1 Amplitude Envelopes for Overlap-Add Synthesis 5
2.2 Linear remapping to epsilon 8
2.3 Calculating both $\sin(\theta)$ and $\cos(\theta)$ with a hybrid technique: table-lookup for α and Taylor expansion for β combine to give an accurate 32-bit result efficiently. 10
2.4 Absolute error as a function of epsilon 11
2.5 Worst-case relative error due to frequency coefficient quantization. 12
2.6 A model of accumulated noise due to rounding at multiplies. 13
2.7 A 500 partial square wave with base frequency 20Hz and partials up to 19980Hz. . 14
2.8 An inexact 1Hz beat frequency generated by synthesizing 10000Hz and 10001Hz signals for two seconds in a single frame. 15

3.1 Sine oscillator operation bubble graph. Multiple operations enclosed in a box are fused into a single instruction in the T0 pipeline. 17
3.2 Amplitude adjustment and sample component summation operation bubble graph. 18
3.3 The bit-level encodings of state variables and intermediates in the sine calculation. 19
3.4 A slice of the three-space defined by measurements of Partials vs. Frame Length vs. Time at a frame length of 256 samples (5.8ms). 20
3.5 As frame length gets very small, initialization overheads dominate. 21

A.1 A single iteration of the oscillator update. 28

List of Tables

3.1 List of operations with descriptions. 16
3.2 Some metrics for comparison of additive synthesis techniques 21

A.1 Descriptions of variables in the annotated code. 28

Chapter 1

Introduction

1.1 Additive Synthesis

Additive synthesis is a signal synthesis technique based on the Fourier Theorem. This theorem states any signal can be decomposed into a set of constituent sine waves, and that the sum of the constituents will reconstitute the original. Additive synthesis is classified as a receiver-based synthesis algorithm, but differs from receiver-based schemes such as subtractive synthesis and sampling in that it is represented in the spectral (frequency) domain rather than the time domain.

There are many benefits in the use of additive synthesis for sound production in computer music applications. These include expressive musical control over fine timbral distinctions, perceptually relevant parameterizations, sample rate independence of timbre description, availability of many analysis techniques, high control bandwidth, and multiple dimensions for resource allocation/optimization [1]. Its use also leverages existing tools and structural manipulation techniques for the domain [2, 3].

The challenge of the additive technique lies in its voracious appetite for separately controllable sinusoidal partials. A single low frequency piano note can require hundreds of time-varying sinusoids for accurate reproduction. Musically effective use of additive synthesis in live performance can require the ability to control many hundreds or even thousands of sinusoidal partials *in real-time.*

Our goal is to provide significantly more real-time partials than are possible using conventional general-purpose hardware architectures. To do so, we develop an engine for real-time additive synthesis implemented on the T0 vector microprocessor.

T0 (for "Torrent-0") [4] is an implementation of the Torrent architecture. This architecture – developed jointly by the University of California at Berkeley and the International Computer Science

Institute (ICSI) – tightly couples a scalar MIPS core to a high-performance vector coprocessor. T0 comprises two vector arithmetic units and a vector load/store unit along with the general-purpose MIPS core, and was designed along with the SPERT neural network and signal processing accelerator board [5], which houses it.

The major features of T0 that drive the design is the vector instruction set architecture (ISA) and the use of fixed-point arithmetic. The explicit parallelism of the vector ISA led us to use parallel recursive oscillators. The 16b fixed-point arithmetic required adapting the recursive oscillators to provide additional accuracy. The modified oscillator is a two-pole filter that maintains frequency precision at a cost of one additional operation per filter sample. The new filter's error properties are expressly imperfect, explicitly matched to use in the context of digital audio rather than general-purpose applications.

We describe the controlling synthesizer software, the control structure for feeding the oscillators, data flow and state management, fast initialization (needed for short synthesis windows), and applicability to related architectures. We present algorithm performance analysis and measurements of the implementation, focusing on how chip features affected algorithm design choices. The technique achieves 608 simultaneous real-time partials (at 44.1KHz) with only a 40MHz clock performing 8 operations per cycle (peak), or about 1.5 cycles per partial per sample.

We also compare our technique to implementations using table-lookup and direct transform-domain methods. Our most basic performance metric is the number of simultaneous, real-time, frequency-variant and amplitude-variant sinusoidal partials. Other metrics where our implementation is successful include its the ability to produce a phase-accurate resynthesis, its ability to independently dose partials into multiple output channels with minimal additional overhead, and in the latency induced by the engine.

1.2 System Framework

An example of a complete additive analysis/synthesis system framework is pictured in Figure 1.1 [6]. It excludes steps for discovering and incorporating stochastic signals into the analysis since our engine does not utilize this information.

Only the steps to the right of the thick dashed line need to be computed in real-time. The concept behind this particular separation is that a set of sound primitives, expressed as sets of overlap-add frames called *timbral prototypes* (detailed in Section 2.1), can be generated off-line via the non-real-time steps as part of the compositional process. Then, at performance time, sets

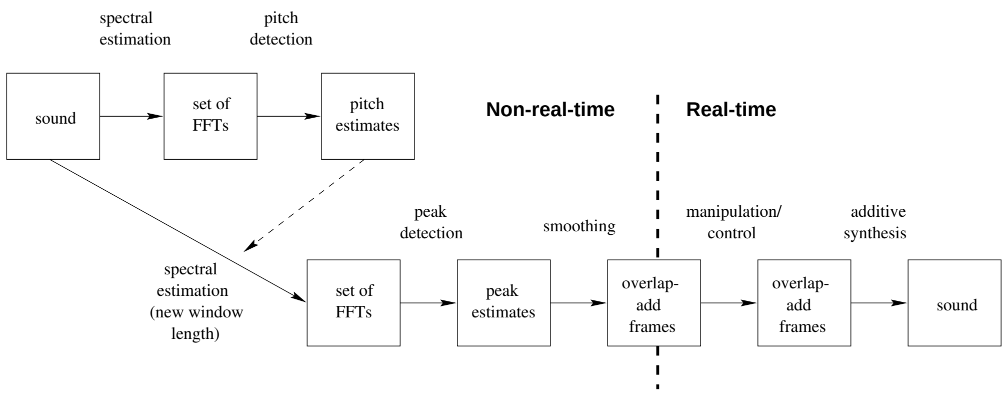

Figure 1.1: An example analysis/synthesis framework including real-time procedures for live performance

of timbral prototypes are loaded into and out of memory according to a score, where they can be manipulated and combined in response to controller input from a performer. The modified frames are then synthesized in real time for subsequent audition. The use of such a paradigm enables additional degrees of freedom in performance than available through, for example, conventional sample-playback-based synthesis.

For this paper, we focus on the final step, that of taking a set of dynamically changing frames and synthesizing them into audio samples. The analysis/resynthesis system we leverage [7] allows the frame manipulation/control and additive synthesis functions to run on separate hardware components, allowing us focus on optimizing the latter independent of the former. Efficient real-time joint implementations are another possibility, but remain as future work.

The rest of the paper is structured as follows. In Chapter 2, we describe the synthesis algorithm and including our changes to a two-pole filter to address the constraints of T0's 16b fixed-point arithmetic. In Chapter 3, we analyze the algorithm performance and provide measurements. In Chapter 4, we discuss programming considerations and some of the possible variations we considered. In Chapter 5, we provide some conclusions, and, finally, in Appendix A, we annotate the inner stripmine loop of the implementation.

Chapter 2

Method Description

This section describes our synthesis algorithm designed for a fixed-point vector microprocessor.

The challenge of using a vector ISA is explicitly managing parallelism. Fortunately, additive synthesis has plenty of inherent data parallelism due to the independence of partial computations. We exploit the natural coarse-grained parallelism by choosing to stripe partials' state variables across the length of the vectors.

The challenge of using moderate-precision arithmetic units (and moderate-precision numeric representations) for recursive oscillators lies in:

- providing sufficient frequency coefficient resolution, and
- dealing with quantization-induced noise effects.

The former is provided by our modification to the standard recursive form; the latter is provided by keeping individual oscillators short-lived in order to exploit the short-term fidelity.

We now provide details of these approaches. The overlap-add approach to windowing addresses the problems with error accumulation inherent in recursive methods. Our two-pole filter includes an ad hoc floating point modification that protects frequency precision across the range of the audible spectrum.

2.1 Interface and Framing

The input expected by our engine is a series of variable-length *overlap-add frames*. A succession of such frames constitute a *timbral prototype*, which is either synthetically designed or derived through a separate analysis phase [8], as shown in Figure 1.1.

Frames consist of the following elements:

- **Frame Header:** a double-precision floating point timestamp denoting the start time of the frame and an integer denoting the number of partials in it.
- **Frame Data:** a list containing the fixed frequency, peak amplitude, and initial phase for each partial in the frame, all in single-precision floating point.

At any instant of time, a timbral prototype is being synthesized as a weighted sum of two constituent frames. Each of the two sets of frame data are synthesized at a constant frequency and phase. Irrespective of their timestamps, successive frames are 50% overlapped with individual amplitude envelopes linearly increasing from zero to the specified peak amplitude value for the first overlapped portion of the frame, and linearly decreasing from this peak back to zero during the second portion of the frame. This is illustrated in Figure 2.1. The two sets of scaled, overlapped frame partials are summed to constitute an output channel.

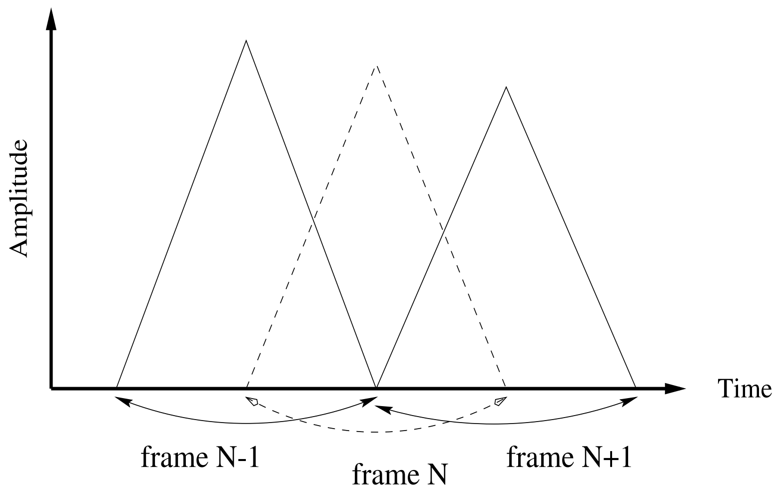

Figure 2.1: Amplitude Envelopes for Overlap-Add Synthesis

An important feature of this approach is that for individual generating oscillators, the frequency, phase, and amplitude remain *constant*. By overlapping and adding successive oscillators with the triangular amplitude envelope, two fixed-frequency, fixed-amplitude sinusoids closely approximate a single varying-frequency, varying-amplitude partial [9].

2.2 Oscillators

There are many ways to generate sinusoids. The most common methods include various recursive techniques, table-lookup, and transform domain methods like those using the inverse FFT [10].

Table lookups require support for a huge number of parallel memory accesses; yet for our vector processor (like for most general-purpose processors), memory references are more expensive than arithmetic. Although transform domain methods are feasible, we chose instead to base our implementation on recursive techniques due to their heavy reliance on explicitly parallel arithmetic with fewer memory accesses. The particular recursive form we utilize is the digital resonator of Smith [11, 12]. Requiring only a single multiply per oscillator, it is computationally less expensive than the waveguide oscillator [13] or the modified coupled form [14]. The general form of the digital resonator, with no damping or initialization impulse function, is:

$$x_n = 2\cos\left(\frac{2\pi f}{f_s}\right)\, x_{n-1} - x_{n-2}$$

where f_s is the sampling frequency and $f \in (0, f_s/2)$ is the desired frequency of oscillation.

We now describe a mapping of this oscillator onto T0 that explicitly matched to the vector ISA and 16-bit fixed-point multiplies. (T0 has native support for most 32b fixed-point operations, but only a 16b$\times$16b $\mapsto$ 32b multiply.)

2.2.1 Recasting the Oscillator Form

To use T0 for audio production, we needed to manage the fixed-point units with enough precision to maintain accuracy across the entire audible frequency range, taking special care to provide sufficient frequency coefficient resolution to account for our ability to distinguish subtle differences in low frequencies. There are two key points: the frequency coefficient multiplication is in the critical path, therefore must be fast. Additionally, though, it must be performed with accuracy across a broader range than a 16b fixed-point representation supplies. Simply implementing the recursive formula presented above with $2\cos(2\pi f/f_s)$ stored as a 16-bit fixed-point unsigned value (with the binary point after two digits in order to encompass $[0, \pi)$) assures sufficient accuracy for the mid and high frequency partials, but doesn't provide enough accuracy for the lows. A key consideration is staying below the experimentally-determined just-noticeable difference (JND) psychoacoustic curve for frequency differentiation [15]. This curve can be used to estimate the accuracy necessary for any encoding of the frequency coefficient for the oscillator's frequency to be correct up to the approximate threshold of human differentiation.

Using the JND curve (or simply by specifying a minimum perceptible musical interval), we can calculate the resolution necessary to maintain relative frequency accuracy. Doing so indicates that the low-frequency components require more precision than higher ones (which is intuitive, since we are calculating *relative* accuracy). To minimize perceived error, then, we remap the frequency representations by employing an internal ad hoc floating-point format for the frequency coefficient. This representation is specifically designed for use with a sixteen-bit multiply for the mantissa and variable right shift for the exponent correction, operations that can be *fused* in the T0 vector arithmetic pipelines as a fixed-point operations and thereby take only a single instruction. The format of this representation includes an unsigned sixteen-bit mantissa, m, and an unsigned eight-bit[1] exponent, e, with the exponent biased so that the actual represented value is $\epsilon = 2^{2-e}m$. Thus, the exponent is also the right shift amount necessary to correct a 16b$\times$16b $\mapsto$ 32b multiply with ϵ as an operand. The two in the exponent allows ϵ to range from 0 to 4 if m is interpreted as a fractional amount. This range is required to support f between zero and the Nyquist frequency (i.e., $\omega \in (0, \pi)$).

To see how we utilize ϵ in a recasting of the filter, recall the recurrence relation of our sine generator:

$$x_n = 2\cos(w)\,x_{n-1} - x_{n-2} \tag{2.1}$$

Note that at low frequency, the coefficient $2\cos(w)$ is very close to two. From this observation, we see that lower frequency oscillators synthesized using the formula (as expressed above) will have less accuracy than higher-frequency oscillators due to the need to explicitly represent the leading ones (the high-order bits) of $2\cos(w)$. Numbers closer to zero benefit from the implicit encoding of leading zeros via a higher exponent. In other words, larger values require bits with larger "significance" (absolute value), forcing the the lowest order bits in the same word to also have higher significance, thus having higher worst-case quantization error.

Therefore, we can more effectively use the bits of the mantissa by reversing this relationship, recasting Eq. 2.1 as:

$$\begin{aligned} x_n &= 2\cos(w)x_{n-1} - x_{n-2} \\ x_n &= 2(1-\epsilon/2)x_{n-1} - x_{n-2} \\ x_n &= 2x_{n-1} - \epsilon x_{n-1} - x_{n-2} \end{aligned} \tag{2.2}$$

[1] Only five bits of exponent are actually necessary; the exponent value is really just a shift amount, and a shift of 33 or greater causes the 32-bit accumulation register to underflow. We've rounded to a byte boundary for convenience.

where

$$\cos w = (1 - \epsilon/2) \tag{2.3}$$

This remapping of number representation in the formula costs one extra arithmetic operation per oscillator. What is achieved with this trade-off, though, is the ability to represent lower frequencies with more significant bits by mapping higher frequencies to representations with less significant digits. In particular, as $2\cos(2\pi f/f_s)$ varies from -2 to 2, we define ϵ to vary from 4 to 0. Figure 2.2 gives a visual depiction of the simple linear transform. Smaller frequency values produce smaller values of ϵ, helping to satisfy our asymmetric accuracy requirements.

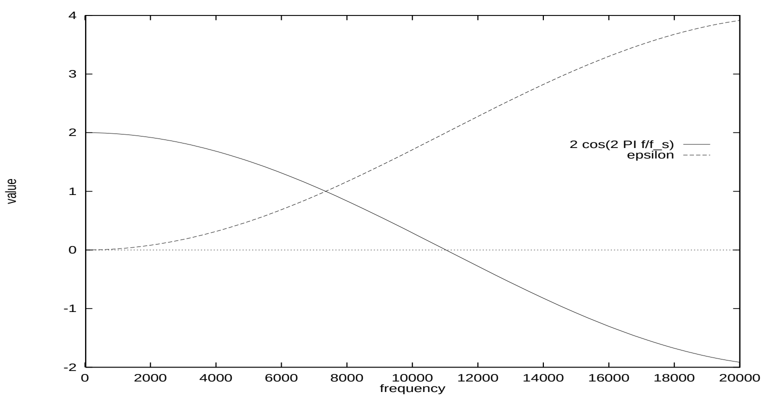

Figure 2.2: Linear remapping to epsilon

2.2.2 Initialization

The resonator oscillator can be initialized to a desired frequency and phase by properly choosing the two state variables x_{-2} and x_{-1}, using function evaluations in place of an initialization forcing function. In our implementation, two sine evaluations are performed per oscillator per frame. The lookup values for a partial with phase p and frequency f are:

$$x_{-1} = \sin\left(p - \frac{2\pi f}{f_s}\right) \qquad \text{and} \qquad x_{-2} = \sin\left(p - \frac{4\pi f}{f_s}\right)$$

This must be accurate down to the low-order bits in a 32-bit fixed point representation, with the binary point set between the third and fourth bit positions in order to support a phase in the range $[0, 2\pi]$.

We implement these initial evaluations by setting

$$\omega = \frac{2\pi f}{f_s} \tag{2.4}$$

and rewriting:

$$\begin{aligned} x_{-1} &= \sin(p-\omega) \\ &= \sin(p)\cos(\omega) - \cos(p)\sin(\omega) \end{aligned}$$

$$\begin{aligned} x_{-2} &= \sin(p-2\omega) \\ &= \sin(p)\cos(2\omega) - \cos(p)\sin(2\omega) \\ &= \sin(p)(2\cos(\omega)^2 - 1) - \cos(p)(2\sin(\omega)\cos(\omega)) \\ &= 2\sin(p)\cos(\omega)^2 - \sin(p) - 2\cos(p)\sin(\omega)\cos(\omega) \\ &= 2\cos(\omega)(\sin(p)\cos(\omega) - \cos(p)\sin(\omega)) - \sin(p) \\ &= 2\cos(\omega)\sin(p-\omega) - \sin(p) \\ &= 2\cos(\omega)x_{-1} - \sin(p) \end{aligned}$$

This recasting allows us to require only the computation of $\sin(p)$, $\cos(p)$, $\sin(\omega)$, and $\cos(\omega)$. The sine and cosine for each argument are determined in tandem in a optimized subroutine that returns both $\sin(\theta)$ and $\cos(\theta)$ for $\theta \in [0, 2\pi]$ to full 32-bit fixed-point precision.

It may seem that this has simply increased the amount of work we need to perform; there are now four trigonometric evaluations rather than three (two initialization sines and the cosine in the recursive form). Though it is unintuitive, this approach turns out to be more efficient by allowing for the judicious sharing of intermediate values in a tandem sine and cosine generation procedure.

The combined $\sin(\theta)$ and $\cos(\theta)$ routine uses a hybrid technique combining table-lookup and Taylor expansion. This keeps both the table size manageable (2048 entries of 32 bits) and the number of terms in the Taylor expansions small (two). It is implemented by separating θ into α and β as shown in Figure 2.3; α is the high-order 11 bits of θ, and β the remaining low-order bits,

α is used in exact (to one LSB) 11-bit $\mapsto$ 32-bit table-lookups, while (guaranteed small) β is used in Taylor expansions:

$$\cos(\beta) \approx 1 - \frac{\beta}{2} \qquad \text{and} \qquad \sin(\beta) \approx \beta - \frac{\beta^3}{6} = \beta\left(1 - \frac{\beta^2}{6}\right)$$

The accuracy of expanding each to only two terms is guaranteed by limiting the size of β to only the low-order 21 bits of θ: the sum of the remaining terms in each expansion sequence, for all β, is less than the LSB. [2]

[2] The proof of this is left an exercise to any curious readers.

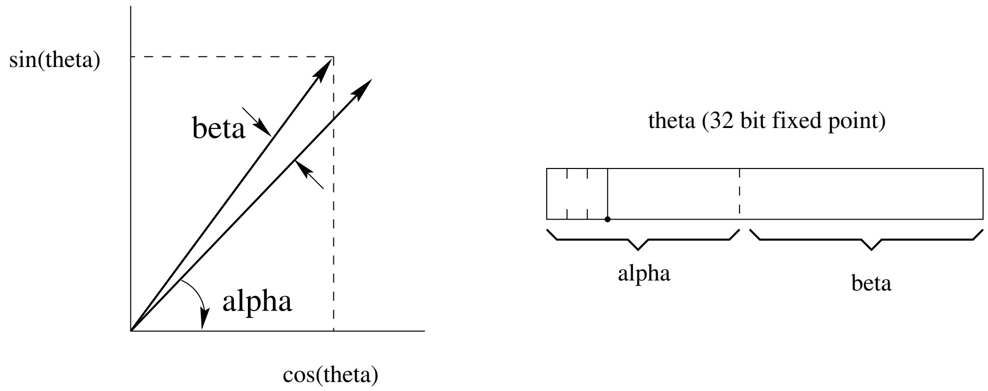

Figure 2.3: Calculating both $\sin(\theta)$ and $\cos(\theta)$ with a hybrid technique: table-lookup for α and Taylor expansion for β combine to give an accurate 32-bit result efficiently.

Finally, α and β are combined using the relationships:

$$\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta) + \cos(\alpha)\sin(\beta)$$

$$\cos(\alpha+\beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$$

2.3 Error Analysis

We follow with an error analysis of our ad hoc floating-point usage in the augmented oscillator form. We discuss worst-case absolute error in frequency resolution, worst-case relative error, the effect of accumulated noise, and the effects of framing.

2.3.1 Absolute Error

Absolute frequency discrimination is based on the ratio of adjacent frequencies. We wish to determine the *maximum* ratio between two adjacent representable numbers. Call these ϵ_1 and ϵ_2, and their corresponding frequencies f_1 and f_2. From Eqs. 2.4 and 2.3,

$$f = \frac{f_s}{2\pi}\cos^{-1}(1-\epsilon/2) \tag{2.5}$$

or

$$f_1/f_2 = \frac{\cos^{-1}(1-\epsilon_1/2)}{\cos^{-1}(1-\epsilon_2/2)} \tag{2.6}$$

The maximum value for ϵ that we choose to represent, 11.11111111111111_2, sets the highest-valued least significant bit of any number in the range to 2^{-14}. We define the exponent, e, to be zero for this location of the binary point, and by design, numbers with this exponent are maintained with the least accuracy. Referring back to Eq. 2.5, we note that this ϵ value coincides with the maximum representable frequency. Because humans perceive pitch as the log of frequency, this achieves a good match of the number representation to the asymmetric accuracy requirements.

Taking any two adjacent numbers in the range, we can compute f_1/f_2 with Eq. 2.6. (Worst-case frequency quantization error can be derived directly from our worst-case epsilon quantization error of $2^{-14}/2 = 2^{-15}$.) The result of determining this ratio for all possible adjacent pairs of epsilon values is illustrated in Figure 2.4. Note that the minimum difference in pitch falls off drastically as e increases, dropping to less than 0.017 cents at $e = 1$, $(\epsilon = 2 - 2^{-15})$. In other words, the middle to high frequencies are represented exceptionally well.

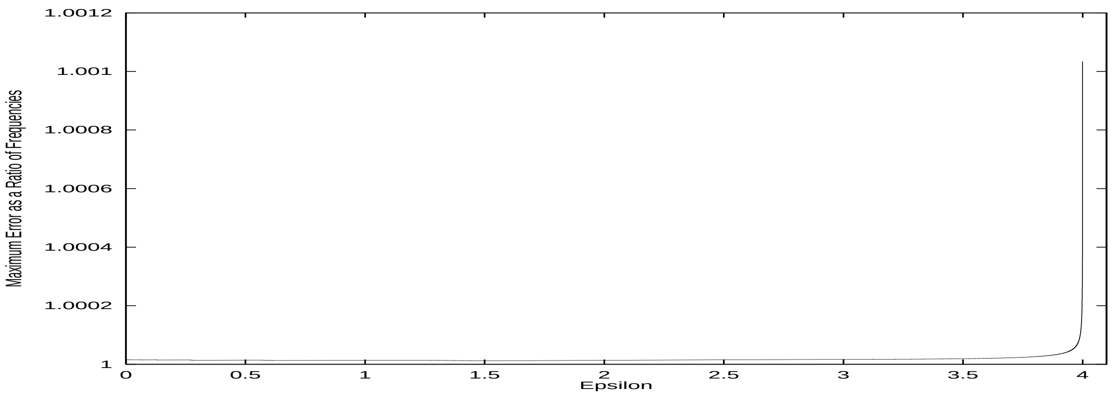

Figure 2.4: Absolute error as a function of epsilon

The function in Eq. 2.6 is maximized for $\epsilon_1 = 4 - 2^{-14}$ and $\epsilon_2 = 4 - 2^{-13}$, where $f_1/f_2 \approx 1.0010337$. This ratio is lower than the minimum frequency ratio humans are able to differentiate, a pitch difference of approximately four to five cents (about 1/25–1/20 of a semitone) [16]. The error is actually less than *two* cents: $\sqrt[600]{2} \approx 1.001156$.

2.3.2 Relative error

Two tones that are meant to have an exact ratio in their frequencies may generate beat frequencies due to epsilon quantization, and this effect – relative error – should be minimized. As might be expected, in addition to supporting tolerance for absolute frequency discrimination, the recast filter also maintains more precise relative frequency. The worst-case error due to relative frequency quantization is show in Figure 2.5.

Fundamentally, more than 16 bits of fixed fractional coefficient are necessary to obtain 1Hz absolute precision across the audible spectrum [17]. Our method maintains reasonable error bounds in 16 bits of mantissa by scaling these bits with the exponent.

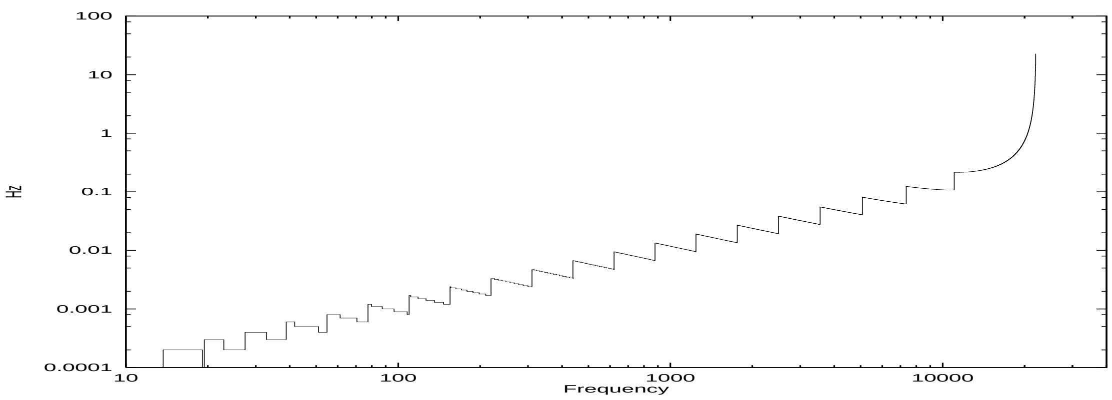

Figure 2.5: Worst-case relative error due to frequency coefficient quantization.

2.3.3 Rounding error

At each iteration of the recursive form, a small error is introduced due to the inexact multiply result representation. Due to the recursion, this error isn't corrected until the next frame boundary. Assuming that errors are uncorrelated and uniformly distributed, this can be modelled as white noise being added to the signal for each multiply. For this case, we plot the highest bit position affected by such noise for various frame lengths in Figure 2.6. It is this effect that requires frame lengths to remain short and forces us to reinitialize the computation at the frame boundaries.

2.3.4 Effects of framing

Care must be taken when dealing with various frame lengths specified as input to the engine due to the recursive nature of the calculation. Though there is no minimum frame time explicit in the current implementation, longer frames allow our synthesis engine more time to amortize the loop setup and input format conversion overhead. This can directly affect the number of simultaneous

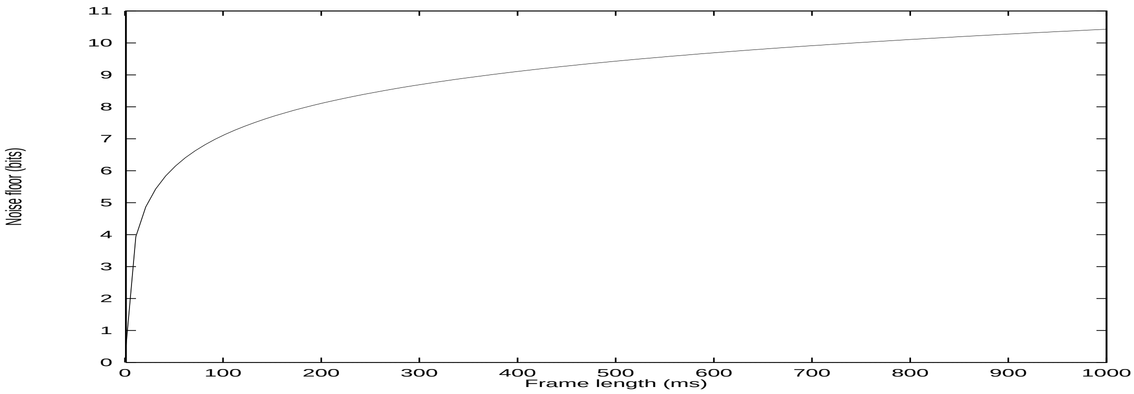

Figure 2.6: A model of accumulated noise due to rounding at multiplies.

real-time partials that can be supported. Values as low as 2.9ms (128 samples) or 5.8ms (256 samples) are often necessary to support fine-grain control for accurate resynthesis of percussive attack sounds, and responsiveness to subtle (short time-scale) user interactions. At the opposite extreme, the input need not conform to any specified maximum frame length in order to guarantee a particular level of performance. One must be maintained internally, though, in order to avoid errors unique to recursive sine generation techniques. Examples of such possible long-term problems include degradation of signal to noise ratio, degradation of long-term phase accuracy, and lack of amplitude stability, all of which can cause audible artifacts [18]. These problems are not intrinsic to recursive techniques; they can be avoided through additional computation, as illustrated in the modified coupled form, a variation of the original coupled form providing ideal numerical behavior [14]. Our use of a non-self-correcting (unstable) but higher-performance oscillator form is made possible by the automatically reinitializing nature of the framing strategy (no oscillator lives longer than two frame times) and by leaving enough headroom in the state variable bit allocations to avoid overflow.

Thus, our strategy implies that the frames must remain short enough to exploit the short-term accuracy, and long enough to amortize the overhead. For the latter we specify an arbitrary minimum, a parameter (defaulting to 5.8ms) that affects the number of achievable real-time partials. As for the former, we can avoid the difficulty internally. Upon detection of a frame length larger than a specified parameter (defaulting to 100ms), the frame is automatically artificially segmented by the engine. This guarantees that necessary accuracy is preserved. Artificial segmentation requires the recomputation of the phase of each partial for the new frame, a cost easily amortized over any frame time long enough to induce it.

Note that the use of frames incurs an unavoidable latency penalty: synthesis cannot begin (let alone be heard) until both overlapped components have been obtained by the engine. The fact that this latency is only one full frame time plus the sample output loop unrolling depth is a benefit over spectral techniques such as the IFFT that require two full frames.

2.4 Example output

Example outputs are illustrated in Figures 2.7 and 2.8. The square wave spans the entire frequency range, is composed of a very large number of partials, and allows a viewer to easily recognize noise artifacts due to its intuitive expected shape. The beating output illustrates relative frequency quantization error and the difficulty of long (ins this case, for illustration, unsegmented) frame lengths. The shape of the first beat is due to it being faded in from time zero (i.e., there is no previous frame)

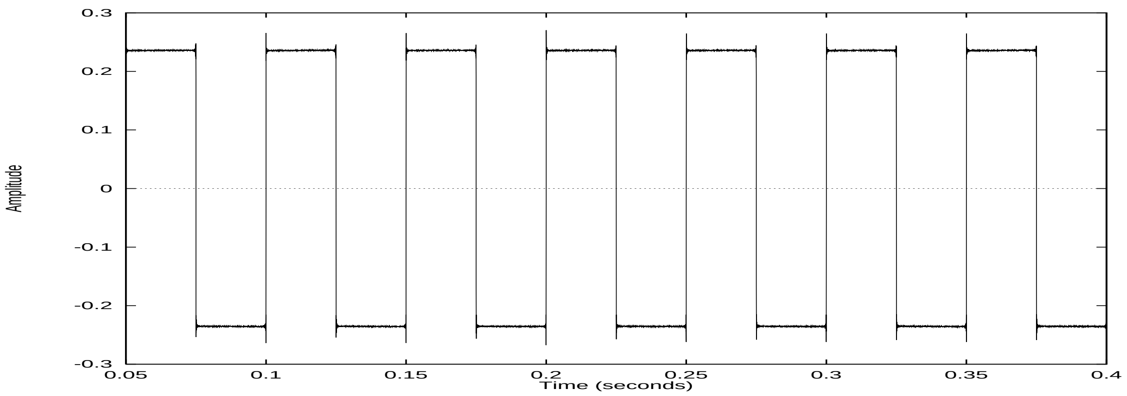

Figure 2.7: A 500 partial square wave with base frequency 20Hz and partials up to 19980Hz.

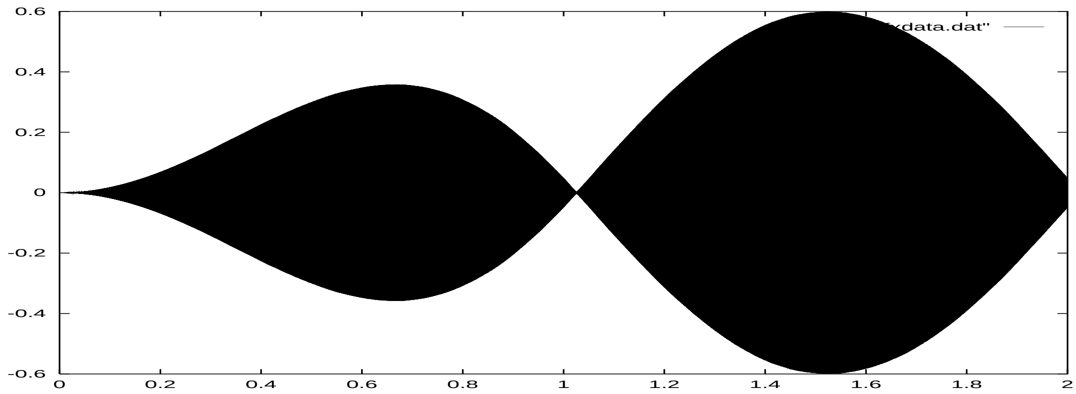

Figure 2.8: An inexact 1Hz beat frequency generated by synthesizing 10000Hz and 10001Hz signals for two seconds in a single frame.

Chapter 3

Performance Analysis

3.1 Analytic

In our implementation, each set of two overlapped sines requires four multiplies and five adds. Using the particular form of the resonator in Eq. 2.2, and the floating point format described in Section 2.2.1, we need four multiplies, two variable shifts, two fused (constant) shifts and adds, and two regular adds. These operations are summarized in Table 3.1.

We use 32-bit fixed-point intermediates for the amplitude, the amplitude delta, and the state variables (x_i, x_{i-1}, x_{i-2}). In order to satisfy T0 alignment requirements, constant shifts are needed to convert them from 32-bit to 16-bit prior to a multiply. This means we need an additional three arithmetic operations beyond those shown in the table (only three because ϵ's mantissa is already 16-bit). One of these shifts can be obtained for "free" from a prior iteration by software pipelining. This leads to a total of $9\frac{1}{4}$ vector arithmetic operations per sinusoid when unrolled four times, the deepest unrolling that we could achieve due to vector register file pressure. (Register allocation trade-offs are discussed in Section 4.)

Data flow graphs for the oscillator, the amplitude adjustment, and output sample component

Operations	Description
$x_n = 2x_{n-1} - \epsilon x_{n-1} + x_{n-2}$	Generate sample
$A_n = A_{n-1} + \Delta A$	Update amplitude
$\text{out}_i = \text{out}_i + A_n \times x_n$	Scale sample, sum into output channel(s)

Table 3.1: List of operations with descriptions.

summation are illustrated in Figures 3.1 and 3.2. State variable encodings are shown in Figure 3.3.

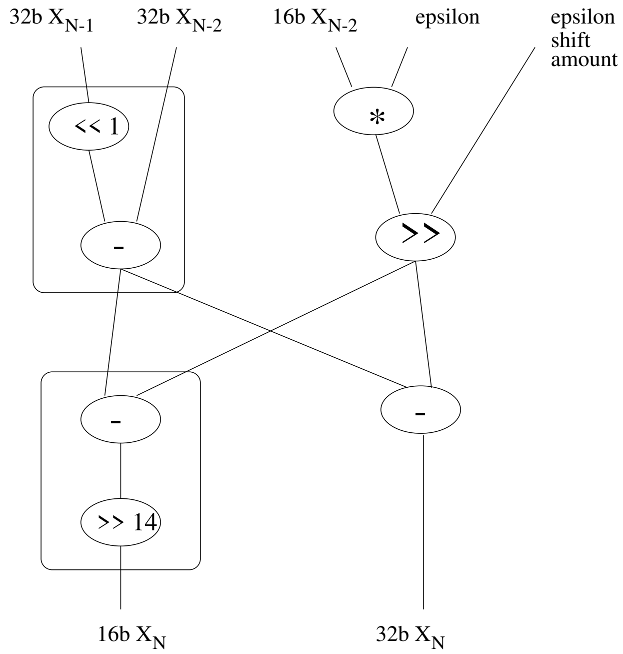

Figure 3.1: Sine oscillator operation bubble graph. Multiple operations enclosed in a box are fused into a single instruction in the T0 pipeline.

Since T0 is a vector microprocessor, all operations specified can be computed n-way data parallel, where n is the vector length. In addition to striping the oscillators across the vector elements for our parallelism, we also (minimally) exploit the available temporal parallelism by pipelining the generation of four samples per iteration through the innermost inner-loop basic block. Increasing this to seven samples is possible, but it performs worse in practice because of the imposition of anti-dependencies due to the sharing of vector registers between the two overlapped-and-added sinusoids in a frame.

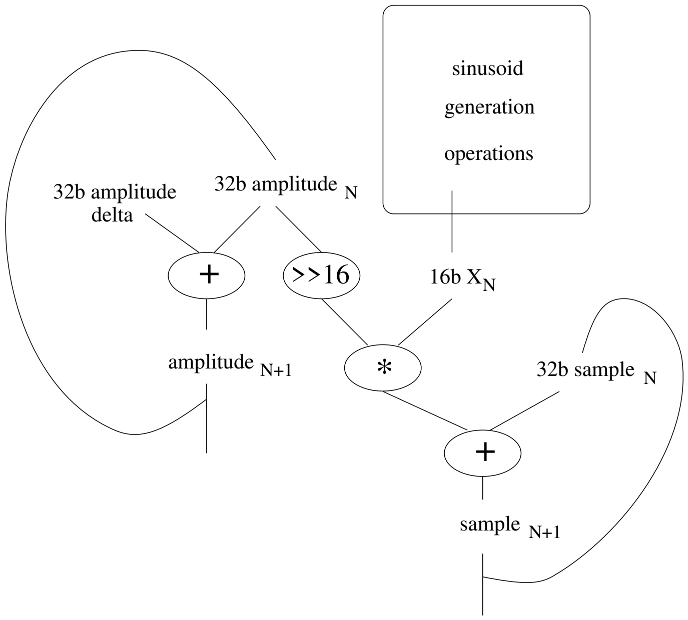

Figure 3.2: Amplitude adjustment and sample component summation operation bubble graph.

Our approach achieves the following best-case performance analysis:

- 32 elements per vector
- 2 sinusoids / partial
- $9\frac{1}{4}$ arithmetic operations / sinusoids
- 2 vector arithmetic operations / 4 cycles

$\Rightarrow$ 37/32 ($\sim$ 1.15) cycles/partial (without overhead).

For a SPERT board with a 40MHz clock rate and at a 44.1kHz sampling rate, this implies that a theoretical maximum of 768 partials can be achieved in real time excluding all overhead.

3.2 Measured

In practice, the above excludes the time necessary for memory access, I/O, control, and scheduling around architectural hazards. The latter is necessary due to register file pressure (anti-dependencies between unrelated stores and loads that happen to share a vector register), short-term

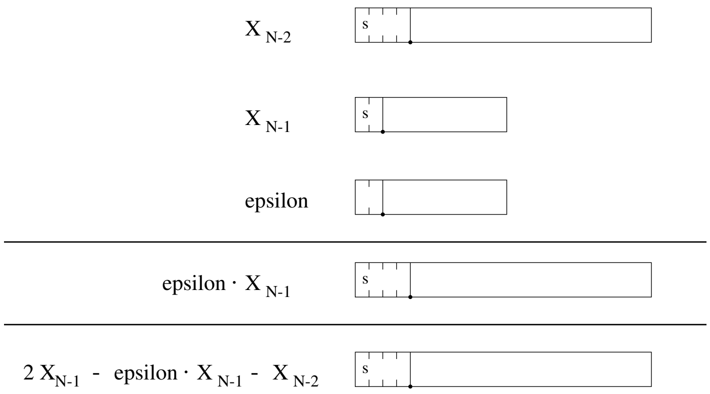

Figure 3.3: The bit-level encodings of state variables and intermediates in the sine calculation.

saturation of the two vector arithmetic units, or loop-setup saturation of the vector memory pipeline. Of these, anti-dependency-based stalls "waste" the most time — we cannot saturate the vector ALUs, which are our critical resource in this particular computation. We also do not saturate the vector memory pipe (VMP) in our implementation using a four times unrolling/software pipelining, which is interesting because we initially expected it to be the critical resource.

Pipeline interlocks due to data dependencies and anti-dependencies could be greatly reduced by scheduling a partial's two constituent oscillators in parallel rather than serializing them. This would necessitate increasing the vector register file from sixteen to nineteen elements.

The current implementation on the prototype accelerator supports up to 608 simultaneous partials sampled at 44.1MHz with frame lengths of 5.8ms or greater. This is illustrated in Figure3.4. These were taken with the host doing the floating point to fixed-point conversion of the frame elements.

As described in Section 2.3, there is a direct trade-off between minimum frame length and maximum achievable simultaneous partials. This is due to the difference in time available to amortize the initialization overhead. The expression of Amdahl's law is illustrated in Figure 3.5.

There is a manifestation of the Heisenberg uncertainty principle for time and frequency such that the frequency content of a single sample point cannot be measured. As time intervals approach zero, their frequency characterizations mean little and frequency-based synthesis becomes less efficient. For miniature frames, then, a time-based or possibly wavelet encoding is more reasonable for efficiency reasons. The SDIF spectral description interchange format [19] supports such combi-

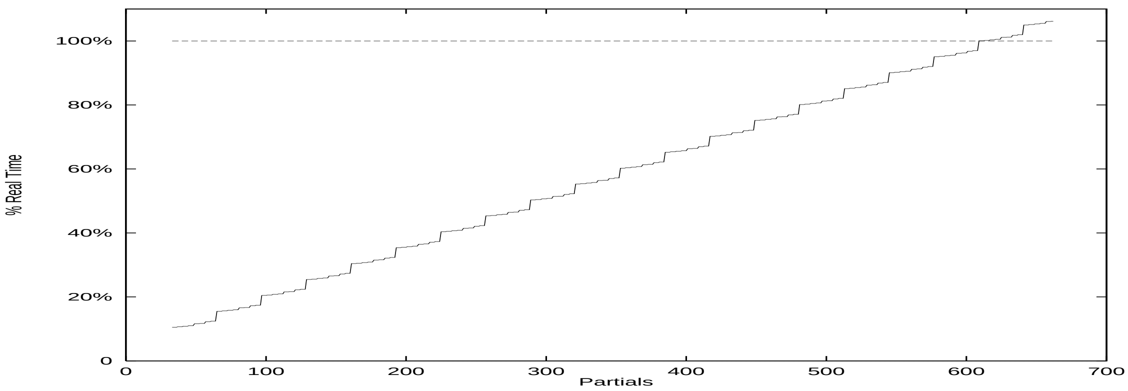

Figure 3.4: A slice of the three-space defined by measurements of Partials vs. Frame Length vs. Time at a frame length of 256 samples (5.8ms).

nations. These hybrids (e.g., combining samples and additive frames in an audio data stream) merit further investigation.

3.3 Other Metrics

In Table 3.2, we discuss metrics other than simultaneous real-time partials. We compare recursive techniques to table-lookup (e.g., a full-custom VLSI implementation [20]) and to spectral-domain techniques (e.g., the IFFT [10]).

Our technique's key advantages over the IFFT are that it can inexpensively separate partials into many independent output channels, that it exhibits lower sample output latency (one frame plus four samples rather than two frames), and that it produces phase-accurate resyntheses. The ability to dose partials into the output channels is useful for implementing frequency-dependent spatial audio (It is a challenge with Inverse FFT method due to the higher per-channel fixed overhead). Phase-accurate output is usually unnecessary for performance, but it is necessary for some analysis methods. A disadvantage is that it does not directly support the efficient incorporation of noise bands into the otherwise deterministic signal, which is easy with the Inverse FFT.

Our technique's key advantage over table-lookup is that it requires only a general-purpose memory subsystem. For a table-lookup synth, though, only a few samples of latency are necessarily induced (due to pipelining of table accesses).

For comparison, we summarize performances of two related implementations: The Inverse

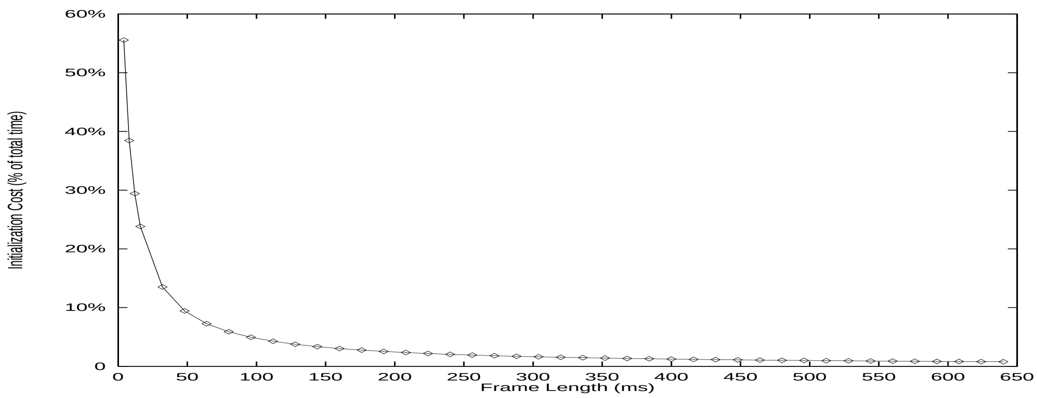

Figure 3.5: As frame length gets very small, initialization overheads dominate.

Metric	**Method**		
	Recursive	Spectral	Table-Lookup
phase-accurate resynthesis	√	×	√
incorporating noise	full cost	reduced cost	full cost
dose partials into independent channels	√	additional cost	√
latency induced	1 frame + few samples	2 frames	few samples

Table 3.2: Some metrics for comparison of additive synthesis techniques

FFT approach running on a 100MHz MIPS R4000-based SGI Indigo Power 2 workstation achieves 350 real-time partials [21]. Running on a R10000-based system, it achieves approximately 1000 partials. A recent prototype custom ASIC implementation [20] achieves 127 real-time partials and a single broadband noise generator using table-lookup in 27mm^2, not including a multiply-add computation per sine wave off-loaded to an external DSP chip.

Chapter 4

Programming Considerations and Variations

4.1 Vector register allocation

In general, register allocation is one of the critical factors in this application. T0 has fifteen available vector registers. We generally reserve one as a temporary, though it can be avoided by selectively clobbering other registers instead. Each frame requires seven state variables[1] (see Table A.1 in the Appendix); of these, three are written out as the result of the calculations, and four are read-only. The question is how to allocate the remaining seven registers.

Possibilities include:

Fully parallel subframes. Thirteen registers (still one less than necessary) are allocated to the subframes, attempting to make them fully independent. The eight read-only registers can then be loaded only once per frame, and only a single register is shared. This leaves only a single register available for output, though, limiting the unrolling depth to one.

Deep Unrolling. Six (or even seven) registers are allocated for output value storage. This allows an unrolling depth of six or seven, but the individual subframe computations are serialized.

Read/Write Sharing. Ten vector registers are used for the frames, allowing the inner basic loop reads and writes to be overlapped, but leaving the read-only registers shared and contended for. This leaves four registers for output value storage, giving an unrolling depth of four. This is the implementation for which we present performance results; it represents the best trade-off in

[1]Note that "output" vector registers are required rather than scalar registers due to our need to keep the vector-to-scalar reduction from 32 elements to 1 element outside the stripmine loop.

balancing contention for the VMP and vector ALUs.

4.2 Amplitude Updates

Another variation allows the removal of the amplitude update from the code (In Figure A.1 of the Appendix, the third to last instruction.). This saves an add operation per oscillator, and thus two per partial, in the inner loop. Instead, a single add and multiply per *sample* are required at frame boundaries, a substantial savings just counting operations. There is a trade-off, of course; moving it outside the loop requires that the two overlapped frames maintain their output values separately at a cost of an extra vector register *per unrolling depth.* These separate buffers can then be independently scaled by a ramp from either $[0\ldots1]$ or $[1\ldots0]$, depending whether it is the incoming or outgoing frame.

Measurements of this variation indicate that the savings of six instructions per iteration doesn't outweigh the register file cost. The lack of available simultaneous storage space for both overlapped subframes' state variables causes false anti-dependencies in memory reads and writes. This causes the burst of VMP saturation that occurs on loop setup to increase, reducing the overlapping of memory-to-register communication and computation. Also, one sample of unrolling is lost due to the required change to register allocation, an additional performance loss.

4.3 I/O

The computation can become I/O bound when streaming dense timbral prototypes through the system:

$$\frac{\frac{12B}{\text{partial}} \times 608 \text{ partials} + 10B}{\text{frame}} \times \frac{1\text{frame}}{2.9\text{ms}} = 2519310\text{Bytes} \approx 20\text{Mbps}.$$

Data compression techniques such as delta coding can be used, but the computation scales in proportion to partial control bandwidth. Thus, timbral prototypes must be loaded into memory and dynamically swapped to and from disk via DMA according to a score rather than allowing them to be streamed in.

As for the numeric representation of the timbral prototypes, parsing the incoming numbers as IEEE floats is expensive. Encoding them as 32-bit integers gives a modest fixed speedup, useful for supporting even smaller minimum-length frames.

The real-time generation tools could be jointly implemented, especially tempting due to T0's prodigious neural network simulation task performance [22] and a current understanding of how to effectively use neural networks for control in live musical performance [23]. In such a case, the input stream will use the fixed-point format naturally.

4.4 Re-implementation performance

Though our technique realizes good performance on the prototype system, it is interesting to extrapolate the performance to a re-implementation of the vector processor. A Torrent chip running at 200MHz in a 0.35μ process (like the current scalar-only MIPS R10000) would support at least 3040 real-time partials. If the re-implementation increased the register file size, the computation could saturate the vector ALUs and supply 3840 partials. Other performance-enhancing changes that would be expected with additional available transistors include an increase in the vector length and number of "lanes." (Lanes are fixed subunits of the vector length unseen by the ISA. The number of lanes is equal to the number of physical parallel arithmetic units in the vector arithmetic units.)

4.5 Application to related architectures

4.5.1 Other Vector Processors

The IRAM project at UCB is implementing a combination vector microprocessor and DRAM [24], the VIRAM. For this architecture, our recursive oscillator approach would be promising due to the vector ISA, but the VIRAM should support IEEE floating point so the modified filter would be unnecessary. Given that it is expected to run at 200-300MHz and that the native floating point saves one operation over our implementation, real-time support for *many* thousands of partials will be feasible. The on-chip multi-megabyte DRAM would provide headroom for storage of many timbral prototypes, making the VIRAM an exciting possibility for use as the embedded processor in an industrial-strength additive synthesizer.

4.5.2 "Multimedia" ISAs

A "multimedia" ISA is just a vector ISA implementing SIMD instructions on small data types packed into existing large registers. Examples of such architectures are those used in the

Intel Pentium (MMX), the Motorola 88110, the HP PA-7100LC, and the Sun UltraSparc (VIS). A multimedia ISA paired with a technique that can do the same work in less bits explicitly exposes extra parallelism. This "extra" parallelism translates into higher peak performance in terms of raw operations per second. Less bits implies lower precision, and thus the underpinnings of our approach are directly applicable to the domain.

Important issues, though, include the small register files of such architectures and the whether a 16b$\times$16b $\mapsto$ 32b multiply is implemented.

Chapter 5

Conclusions

We have presented a detailed description of our implementation of an additive synthesis engine on the T0 vector microprocessor. It achieves 608 frequency-accurate partials (at 44.1KHz) in real time with only fixed-point hardware and at the "meager" clock rate of 40MHz. This is about 1.5 cycles per partial per sample. Our implementation is particularly novel in that it has the ability to produce a phase-accurate resynthesis, can independently dose partials into multiple output channels with minimal additional overhead, and has reduced latency in comparison to the IFFT.

Future work includes a joint implementation with neural-network-based control and manipulation code, and porting to related architectures. We'd also like to qualify how any remaining absolute frequency quantization errors effect the perception of various outputs, whether in distracting (or possibly interesting!) ways.

Appendix A

Annotated Inner Stripmine Loop

T0 uses a novel scheme to control the configuration of the vector arithmetic pipelines. Scalar registers are loaded at run-time with flags and fields specifying a particular configuration of the vector arithmetic pipeline. These *vector control registers* are then associated with a vector instruction of the form

```
fxadd.vv vv1,vv2,vv3,t1
```

where vv1, vv2, and vv3 are vector registers and t1 is the scalar. Upon execution, the dataflow and operations performed on vv1, vv2, and vv3 is determined by both the operation mnemonic *and* the contents of t1. Parameters that can be set via control registers include left and right shift amounts, various clipping and rounding modes, and conditional moves into the result vector register.

Figure A.1 is the code fragment that implements a single iteration of the current implementation's inner stripmine loop, in this case for the sinusoid in the current frame that is being faded in. Its variables are summarized in Table A.1. In the table, the suffix '_v" implies a vector register, "_r" a scalar register, "_fxctrl_r" a control register for a vector instruction. The first instruction is unnecessary in later iterations, since this same result can be obtained from the previous iteration's fifth instruction (the `fxsub.vv`). Pipelining this code four iterations deep obtains the $9\frac{1}{4}$ operations per frame figure quoted in the performance analysis.

Variable	Description
x_v, xm1_v, xm2_v	32b recursion state
eps_v	epsilon unsigned mantissa
eps_exp_v	epsilon shift amount
amp_v	32b (fading) amplitude
ampDelta_v	32b amplitude increment value
sample1_v	output (sample components)
mult_fxctrl_r	second operand is unsigned
left1_fxctrl_r	shift first operand left one
sixteen_r	constant 16
res_v	temporary

Table A.1: Descriptions of variables in the annotated code.

```
# convert 32bit -> 16 to align for mult
srav.vs         x_v, xm1_v, sixteen_r
# gives x_n-1 * eps (unshifted)
fxmul.vv        x_v, x_v, eps_v, mult_fxctrl_r
# shift to get fully-accurate x_n-1 * eps
srav.vv         x_v, x_v, eps_exp_v
# multiplies (renamed) x_n-1 by 2 and adds it to x_n-2
fxadd.vv        x_v, x_v, xm2_v, left1_fxctrl_r
# gives 16b resulting x_n; saved for use in next iteration
fxsub.vv        xm2_v, xm1_v, x_v, add_fxctrl_r
# shift amplitude to 16b for multiply alignment
fxadd.vv        res_v, amp_v, ampDelta_v, 0
# get 32b resulting x_n
fxsub.vv        x_v, xm1_v, x_v, left1_fxctrl_r
# update amplitude for next sample
add.vv        amp_v, amp_v, ampDelta_v
# gives final output sample contribution from this partial
fxmul.vv        res_v, xm2_v, res_v, mult_fxctrl_r
# sum it into running total
add.vv        sample1_v, res_v, sample1_v
```

Figure A.1: A single iteration of the oscillator update.

Bibliography

[1] P. Cook, R. Bargar, X. Serra, and A. Freed, "Creating and Manipulating Sound to Enhance Computer Graphics," *SIGCOMM tutorial session*, 1996.

[2] A. Freed, "Bring Your Own Control Additive Synthesis," *International Computer Music Conference*, 1995.

[3] Center for New Music and Audio Technologies, "CNMAT Additive Synthesis Toolkit (CAST) project." http://cnmat.cnmat.berkeley.edu/CAST/, 1996.

[4] K. Asanovic, B. Kingsbury, B. Irissou, J. Beck, and J. Wawrzynek, "T0: A Single-Chip Vector Microprocessor with Reconfigurable Pipelines," *Proceedings 22nd European Solid-State Circuits Conference*, September 1996.

[5] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan, "SPERT-II: A Vector Microprocessor System," *IEEE Computer*, vol. 29, pp. 79–86, March 1996.

[6] M. Wright. CNMAT studio documentation, 1996.

[7] A. Freed, "Codevelopment of User Interface, Control and Digital Signal Processing with the HTM Environment," *Proceedings of The International Conference on Signal Processing Applications and Technology*, 1994.

[8] P. Depalle, G. Garcia, and X. Rodet, "Tracking of Partials for Additive Sound Synthesis Using Hidden Markov Models," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pp. 225–8, April 1993.

[9] M. Goodwin and A. Kogon, "Overlap-Add Synthesis of Nonstationary Sinusoids," *Proc. Int. Computer Music Conf.*, 1995.

[10] A. Freed, X. Rodet, and P. Depalle, "Synthesis and Control of Hundreds of Sinusoidal Partials on a Desktop Computer without Custom Hardware," in *Proceedings of ICSPAT*, (Boston. MA), DSP Associates, 1993.

[11] J. Smith and J. Angell, "A Constant-Gain Digital Resonator Tuned by a Single Coefficient," *Computer Music Journal*, no. 4, 1982.

[12] J. Wawrzynek and C. Mead, "A VLSI architecture for sound synthesis," in *VLSI Signal Processing: A Bit-Serial Approach* (Denyer and Renshaw, eds.), Reading: Addison Wesley, 1985.

[13] J. Smith and P. Cook, "The Second-Order Digital Waveguide Oscillator," *Proc. Int. Conf. Computer Music*, 1992.

[14] J. Gorden and J. Smith, "A Sine Generation Algorithm for VLSI Applications," *Proc. Int. Conf. Computer Music*, 1985.

[15] E. F. Evans, "Basic Physics and Psychophysics of Sound, Functional Anatomy of the Auditory System, Functions of the Auditory System," in *The Senses* (H. B. Barlow and J. D. Mollon, eds.), Cambridge University Press, 1982.

[16] J. O. Pickles, *An Introduction to the Pysiology of Hearing*. Academic Press: Orlando, Florida, 1982.

[17] J. Wawrzynek, *VLSI Concurrent Computation for Music Synthesis*. PhD thesis, California Institute of Technology, 1987.

[18] A. Freed. Personal Communication, 1995.

[19] Center for New Music and Audio Technologies, "SDIF: Spectral Description Interface Format." http://cnmat.cnmat.berkeley.edu/SDIF/, 1997.

[20] A. Honghton, A. Fisher, and T. Malet, "An ASIC for Digital Additive Sine-wave Synthesis," *Computer Music Journal*, vol. 19, no. 3, pp. 26–31, 1995.

[21] X. Rodet and P. Depalle, "A New Additive Synthesis Method Using Inverse Fourier Transform and Spectral Envelopes," *Int. Computer Music Conf.*, 1992.

[22] J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan, “SPERT-II: A Vector Microprocessor System and its Application to Large Problems in Backpropagation Training,” *Proceedings NIPS ’95*, 1995.

[23] M. Lee and D. Wessel, “Soft Computing for Real-Time Control of Musical Processes,” in *1995 IEEE International Conference on Systems, Man and Cybernetics*, vol. 3, pp. 2748–53, 1995.

[24] Patterson, D., et. al., “A Case for Intelligent DRAM: IRAM,” *IEEE Micro*, April 1997.