The Case for User-Level Preemptive Scheduling to Support Multi-Rate Audio Applications for Multi-Core Processors

by Rimas Avizienis

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Krste Asanovic
Research Advisor

(Date)

* * * * * * *

Professor David Wessel
Second Reader

(Date)

Chapter 1

Introduction

As general-purpose computers have become ever more powerful, their use by musicians, multimedia artists and sound engineers to synthesize and process audio in real time has steadily increased. The computational capabilities of modern microprocessors are such that audio signal processing tasks that used to require a rack full of dedicated DSP based audio gear can now be performed by software running on commodity PCs. However, operating system support for audio applications with soft real-time constraints has not kept pace with the widespread adoption of multi-core processors. To compound the problem, audio processing functionality is often packaged into reusable software modules (called ”plugins”) that can be instantiated within audio “host applications” to extend their functionality. Since current audio plugin APIs lack any explicit support for parallelism, the only way plugins can perform their processing tasks in parallel on multiple cores is by spawning additional OS threads. However, a host application has no way of knowing about or managing the scheduling of additional threads that plugins may have been spawned, and the OS lacks sufficient information to intelligently schedule these threads. A similar problem is faced by all developers of parallel programs composed of multiple parallel libraries: because individual libraries are unaware of each other's existence, they may collectively spawn more threads than there are physical cores in a system (a condition known as “oversubscription”) which can lead to sub-optimal and unpredictable performance. The real-time constraints of audio applications make solving the composability problem even more challenging. These factors combined make it very difficult for developers to write audio applications and plugins that can efficiently exploit the

compute resources available in modern CPUs.

To complicate matters further, some audio processing algorithms work by performing computation on an incoming audio stream at multiple rates concurrently. Such algorithms can often be most easily implemented by creating multiple threads that preempt each other as necessary. However, current audio plugin APIs dictate that plugins should perform their time sensitive computations within the context of a single thread. A host application calls a plugin's "execute" function when a new frame of data is ready for processing, and this function is expected to complete in a constant amount of time every time it is invoked. To implement a multi-rate algorithm in a way that is compatible with this execution model, a programmer must partition processing tasks operating at lower frame rates into multiple subtasks that are executed at the highest frame rate. This partitioning is difficult to perform in a load balanced manner and generally precludes the use of external libraries to implement processing tasks. We illustrate this point through a case study of the implementation of non-uniform partitioned convolution (a multi-rate algorithm) using both scheduling paradigms: multi-threaded preemptive, and single-threaded cooperative. The preemptive version was easier to code and is more flexible than the cooperative version, resulting in better performance on a wider variety of processing tasks. However, since current audio plugin APIs don't support preemption (or multi-threaded signal processing routines in general), it isn't possible for applications to host plugins that employ preemption in a way that guarantees deterministic behavior.

We believe that the best way to address these issues is to adopt a two-level scheduling model, where the OS scheduler gang-schedules all of an application's threads and a user-level scheduler (with support for preemption) makes decisions about how to map processing tasks onto those threads. Plugin APIs must be extended so as to allow multiple user-level schedulers within an application to be composed hierarchically, and to support preemptive programming styles. In the following sections, we first provide background on how audio I/O devices interact with operating systems (specifically Linux), and how operating systems interact with applications that have real-time constraints. Next, we describe and evaluate the performance of two implementations of non-uniform partitioned convolution that use different scheduling paradigms. Finally, we present our conclusions and discuss related work.

1.1 Collaboration, Previous Publications, and Funding

This thesis is a result of a collaborative group project. Eric Battenberg implemented the preemptive version, and I implemented the cooperative version of the real-time non-uniform partitioned convolution algorithm described and evaluated in the case study portion of this thesis. Some of the figures and content in this thesis are adapted from a previous publication: "Implementing Real-Time Partitioned Convolution Algorithms on Conventional Operating Systems" [2] from DAFx-11.

This work was supported in part by funding from Microsoft (Award #024263) and Intel (Award #024894, equipment donations) and by matching funding from U.C. Discovery (Award #DIG07-10227).

Chapter 2

Background

In this chapter, we review some basic concepts related to digital audio and describe the interfaces between audio hardware I/O devices, operating systems, and audio applications.

2.1 Digital Audio Basics

In the physical world, what we experience as sound corresponds to variations in air pressure, or mechanical sound pressure "waves". Our ears respond to such pressure waves by sending a series of neural impulses down the auditory nerve to the brain, causing us to perceive sound. The ear's response to sound waves is not strictly linear - the neural impulses generated by the inner ear actually represent a processed version of the incident sound. This means that a listener may perceive two different sounds as equivalent. "Lossy" audio compression algorithms (such as MP3 [17]) take advantage of this property by applying psychoacoustic models to determine which components of an audio signal are perceptually significant and encoding them efficiently. Such algorithms can achieve a tenfold reduction in the space required to represent an audio signal while introducing a minimal amount of perceptible distortion.

Microphones are transducers that convert mechanical vibrations (sounds) into electrical signals. A microphone is connected to a "pre-amplifier" that outputs a voltage proportional to the magnitude of the vibrations experienced by the microphone's sensing element (called a capsule). This is how audio signals are represented in the analog domain.

Uncompressed digital audio signals are represented as pulse code modulation (PCM) encoded streams. An analog audio signal's voltage is measured (sampled) periodically to produce a stream of digital values (samples). Each sample is a signed integer with a fixed width (called the bit-depth) that is a quantized approximation of the magnitude of the analog signal's voltage. The sampling rate determines the highest frequency that a discrete time digital signal can encode. This is known as the Nyquist frequency and is equal to half the sampling frequency. An analog signal must be filtered to remove frequencies above this threshold before being sampled in order to avoid aliasing in the resulting digital signal. The dynamic range of a signal encoding is defined as the ratio between the largest and smallest magnitudes the encoding can represent. For PCM encoded audio, the bit depth determines the dynamic range of the signal: each bit contributes 6 decibels (dB) to the dynamic range. Commonly used sampling rates and bit depths are 44.1 kHz/16 bits (CD quality) and 96 kHz/24 bits (DVD quality), though newer digital media formats (such as Blu-Ray) support sampling rates of up to 192 kHz. Signal processing algorithms implemented on modern CPUs generally employ floating point arithmetic, and audio samples are converted between integer and floating point numbers (in the range -1.0 – 1.0) as necessary.

For efficiency reasons, general-purpose computers process digital audio in batches of samples called frames, where a frame consists of some number (known as the I/O vector size) of samples per audio channel. I/O latency is defined as the amount of time that elapses between the arrival of a signal at a device's input and the appearance of a processed version of the signal at a device's output. The audio I/O vector size imposes a lower bound on a system's achievable I/O latency equal to double the frame period, in order to allow the computer an entire frame period to process a frame's worth of audio.

Interactive audio applications demand minimal I/O latency. In some situations, such as live sound reinforcement, any perceptible latency is unacceptable. In practice, latencies below 10 milliseconds are generally considered acceptable when using general-purpose computers to process audio in real-time, though standalone DSP based audio signal processors can achieve latencies on the order of tens to hundreds of microseconds. We aim to achieve latencies on the order of 1.5 – 3 milliseconds, which necessitate using I/O vector sizes of 32 or 64 samples at 44.1 kHz. As is often the case in digital systems, there is a tradeoff between efficiency (bandwidth) and latency (responsiveness).

2.2 Gestural Interfaces

The MIDI [13] (Musical Instrument Digital Interface) protocol provides a way to connect gestural interfaces (called "controllers" in this context) to other devices (e.g. computers, standalone synthesizers). The most common type of MIDI controller is a piano-style musical keyboard. Striking one of its keys causes it to transmit a MIDI "Note On" message identifying the key and indicating the velocity with which it was struck. When the key is released, the controller sends a complementary "Note Off" message. A computer equipped with a MIDI interface can send and receive MIDI messages. MIDI controllers are typically used to trigger other devices to produce sounds. While MIDI is an antiquated protocol (the MIDI specification was first published in 1983) and the physical transport associated with the MIDI protocol (a 31.25 kilobaud serial link) is rapidly disappearing, the MIDI protocol is still often used as a way to encode audio-related events.

Open Sound Control [22] (OSC) is another, more recent protocol for communication between gestural controllers, computers and other multimedia devices. OSC support has been integrated into a wide range of hardware and software products. OSC has numerous advantages over MIDI: it supports precise temporal semantics through the use of time tags and atomic operations, its messages are formatted in a human readable manner, and it can be used over both local and wide area networks. Open-source implementations of the OSC protocol are available for embedded microcontroller platforms (such as the Arduino) as well as conventional operating systems (Windows, Linux, MacOS).

2.3 Audio Input/Output Hardware

Audio I/O interfaces provide a host computer with audio inputs and outputs which can be either analog or digital in nature. To handle analog signals, an audio interface must include analog to digital and digital to analog converters (ADCs and DACs). Standard protocols (such as S/PDIF, ADAT, and MADI [19]) which operate over electrical or optical transports are used to transfer audio data between devices in digital form. These protocols include methods for performing clock synchronization between digital audio devices. Clock synchronization is necessary because two devices nominally operating at the same sampling frequency are not perfectly synchronized. Internally, devices employ crystal oscillators to generate the clock signals required by ADCs and

DACs. Miniscule differences in the oscillation frequencies of individual crystals accumulate over time and cause the clock signals derived from them to "drift" with respect to one another. To avoid this problem, one audio device in a system is designated as the clock "master" and the rest as "slaves." Slave devices use phase locked loops (PLLs) or voltage controlled oscillators (VCOs) to generate clock signals that are synchronized with the master clock.

Most modern laptops and motherboards include built-in analog stereo input and output devices. Users that require more channels or better fidelity can choose from a plethora of third-party audio interfaces. These devices connect to host machines either directly through the PCI/PCIe bus or via USB, FireWire or Ethernet interfaces. Audio data is transferred between audio interfaces and a host computer in frames, and audio devices usually support a range of I/O vector sizes, usually between 32 – 4096 samples. Audio interfaces typically notify the host that a new frame of data is ready by asserting an interrupt, though polling can be used instead for non latency sensitive applications. Audio interrupts occur at a constant rate, with a period determined by the I/O vector size.

2.4 Audio Hardware/Operating System Interface

The OS reacts to an audio interrupt by executing an interrupt service routine (ISR) in the audio device driver. The ISR is responsible for copying data between the device and buffers in the device driver, and notifying the OS to mark threads blocked on audio I/O as runnable. The OS scheduler then decides when and for how long these threads get to execute. Interrupt latency is defined as the delay between the assertion of an interrupt and the execution of the ISR, and scheduling latency is defined as the delay between the execution of the ISR and the resumption of a blocked audio I/O thread. These latencies reduce the time available for an application to compute a frame of output data without missing deadlines and causing dropouts in the output audio signal.

2.5 Operating System/Application Interface

Applications typically register a callback function with the OS to receive notifications about the arrival of audio data. The details of how this is done vary between OSs, however the result is the same: the callback function is invoked whenever a new frame of audio data is ready. The

callback function executes in the context of a dedicated high priority thread (the audio I/O thread) so as to minimize its scheduling latency. The most efficient way to move audio data between the OS kernel and a user-space application is to map the OS audio I/O buffers into a region of the application's address space. An application can then copy audio data between OS and application buffers without making any system calls, which would require additional kernel crossings and would further reduce the time available for useful computation.

2.6 Audio Plugins

Audio processing functionality is commonly packaged into reusable software modules known as "plugins," implemented as dynamic libraries. By adopting a standard API, authors of audio plugins enable their plugins to be used within multiple "host" applications. Examples of popular audio plugin APIs are LADSPA (Linux Audio Developer's Simple Plugin API), Steinberg's VST (Virtual Studio Technology) and Apple's Audio Units. Plugins can be broadly classified into two categories: those which only produce output (commonly referred to as "virtual instruments") and those which process inputs to produce outputs (commonly called "effects processors"). There are many types of virtual instruments, examples include emulations of analog synthesizers, physical models of acoustic instruments, and samplers. Examples of effects processors include filters, reverberators, compressors, noise-gates and harmonizers.

Plugin APIs provide mechanisms for host applications to instantiate plugins, adjust their parameters, manipulate their input and output buffers, and trigger their internal signal processing routines. A major shortcoming of current plugin APIs is the absence of any explicit support for parallelism. Current plugin APIs define an "execute" function which a host application invokes, causing the plugin to generate a frame of output. This function is required to execute in a constant length of time each time it is invoked, and is forbidden from making system calls or otherwise engaging in behavior that would lead to non-deterministic execution times. This execution model makes it all but impossible for plugin authors to write plugins that can use multiple cores to perform their signal processing computations in parallel.

2.7 Audio Applications

There exist many types of audio applications - three major categories are digital audio workstations (DAWs), sequencers, and visual patching languages. DAW software is used to record, manipulate, and play back digital audio streams (also called tracks). Examples of DAW applications include Digidesign's ProTools and Apple's Logic. DAWs emulate the behaviour of analog recording studios, which usually include a multi-track recorder, a mixing console and a rack of signal processing gear. DAWs allow users to simultaneously record and/or play back multiple audio tracks while processing them by applying effects, and to "mix" multi-channel recordings down to a stereo or surround (i.e. 5.1, 7.1) formats suitable for playback on consumer audio equipment.

Sequencers are another related category of audio applications. Instead of manipulating pre-recorded audio tracks, sequencers operate on audio represented as sequences of events. These events are most commonly encoded as MIDI commands and can be used to trigger the generation of sounds by virtual instruments and to adjust the parameters of virtual instruments and effects processors. Sequences can be represented visually as scores or "piano rolls," and sequencers enable musicians to compose and arrange songs by manipulating "loops" of audio by cutting and pasting, modifying tempo and pitch, cross-fading, and so on. Examples of popular sequencers include Ableton Live and Apple's Garage Band. The distinction between DAWs and sequencers is becoming increasingly blurred, as most modern DAWs provide some degree of sequencer functionality and most sequencers include mechanisms to playback and record digital audio tracks.

Another category of audio applications are visual patching languages, the most prominent examples of which are the open-source package PureData (PD) and Cycling 74's Max/MSP. These are more accurately described as programming languages as opposed to applications with well defined functionality. They provide a programming environment in which visual representations of signal processing objects are connected by virtual wires to construct "patches." The term patch is a relic from the early days of electronic music, when patch cables were used to connect oscillators and filters to construct synthesizers. Patches can be packaged into modules (called "sub-patches") which can then be instantiated within other patches in the same way as other objects. These environments are very versatile, as users can also write their own objects (called "externals" in this context) in C/C++ to extend the functionality of the system.

Current audio applications have limited support for parallelism at a very coarse grain. For ex-

ample, in most modern DAWs a user can statically partition the execution of independent "chains" of plugins across multiple threads. The most recent version of Max/MSP allows multiple, independent top-level patches to execute concurrently in separate threads. However, there are currently no well-defined mechanisms to enable a single instance of a patch or plugin to use multiple cores to perform its internal processing tasks in parallel. The burden of partitioning processing tasks across multiple cores thus rests with the user, resulting in underutilization of processing resources and/or significant effort spent tuning individual patches or DAW configurations for particular platforms and use cases.

2.8 Audio Programming under Linux

The mainline Linux kernel provides two scheduling policies to support applications with soft real-time constraints: these are SCHED_FIFO (first-in first-out) and SCHED_RR (round robin). A third scheduling policy (SCHED_OTHER) is used for the rest of the threads in the system. Every thread in the system is assigned one of these policies, as well as a static priority in the range 0 – 99. Threads using the SCHED_FIFO and SCHED_RR policies are scheduled before any SCHED_OTHER threads. All SCHED_FIFO threads at a given priority level are scheduled in FIFO order and allowed to run until they block, or until another SCHED_FIFO thread with a higher priority becomes runnable (in which case it will preempt a currently running thread if necessary). Lower priority threads aren't scheduled until all higher priority threads are blocked. SCHED_RR threads are scheduled similarly, except that threads with the same priority are scheduled round-robin using a default scheduling quantum of 10ms. Audio applications do their time-sensitive computation using SCHED_FIFO threads. A kernel parameter sets a limit on the CPU time (expressed as a percentage of total CPU time) allotted by the system for SCHED_FIFO and SCHED_RR threads - by default it is set to 95%. This prevents misbehaving soft real-time threads from hanging the system.

The audio framework in Linux is called ALSA: the Advanced Linux Sound Architecture. It consists of kernel drivers for a wide variety of audio hardware devices, as well as a user-space library and API for application developers. The ALSA API enables a variety of programming styles, largely to support legacy code that uses the older OSS (Open Sound System) audio API. Unfortunately, as is the case with many open-source projects, the ALSA API is poorly documented.

Discovering the best way to use ALSA to implement a full-duplex, low-latency real-time signal processing application required reading a significant amount of source code and a good bit of experimentation. For this reason, many developers choose to use a cross-platform solution such as the JACK [11] audio server daemon to implement communication between applications and audio devices. JACK was designed with low-latency, real-time applications in mind and it does a good job of supporting such applications while introducing a minimal amount of overhead. For most developers, the slight overhead associated with using JACK outweighs the additional complexity of interfacing with ALSA directly. We opted to use the ALSA API instead of JACK both to improve performance, and to make it easier to understand the scheduling behavior of the system.

Chapter 3

Convolution Algorithms

Convolution is a mathematical operation that appears frequently in the field of digital signal processing. It describes the behavior of finite impulse response (FIR) filters, which convolve an input signal with an impulse response to produce a filtered output signal. In audio signal processing, FIR filtering is often used to simulate reverberation. When sound is produced in an enclosed space, a listener in that space hears the original sound, followed by a series of decaying echoes caused by the sound pressure waves reflecting off of the surfaces in the room. The timing and amplitude of these reflections is called the room response, or equivalently the impulse response of the room. Convolving a "dry" sound source (one recorded in an anechoic environment) with a room response results in an output signal which approximates how that sound would be perceived by a listener in the room. The duration of the room response (how long it takes for sound to decay to an imperceptible level) is related to the size and shape of a room as well as the acoustic properties of its surfaces. The room response of large spaces can last for ten or more seconds, translating to an impulse response that is almost 500k samples long (at 44.1 kHz sampling rate). Convolving such an impulse response with an input audio signal in real-time with low latency is a very computationally intensive task. In the following sections, we describe several methods for computing the convolution of two signals.

3.1 Convolution in the Time Domain

Convolution is defined as

$$\mathbf{y} = \mathbf{x} * \mathbf{h}$$
$$\mathbf{y}[n] = \sum_{k=0}^{N-1} \mathbf{x}[k]\mathbf{h}[n-k]$$

where two N point sequences $\mathbf{x}$ (an input signal) and $\mathbf{h}$ (an impulse response) are convolved to produce the $2N-1$ point output sequence $\mathbf{y}$. Computing the convolution of two signals by direct application of the above formula has no inherent latency (results can be computed on a sample by sample basis) and has a computational cost of $O(N)$ operations per output sample. Since its computational cost increases linearly with the length of the impulse response, the direct method is not a practical way to perform convolution with very long impulse responses.

3.2 Convolution in the Frequency Domain

Convolution can also be computed by multiplying the frequency domain representations of two signals, since for discrete time sequences, cyclic convolution in the time domain is equivalent to multiplication in the frequency domain [16]. The discrete Fourier transform (DFT) coefficients are a frequency domain representation (spectrum) of a time domain signal. The forward DFT of a sequence $\mathbf{x}$ of length N is defined as

$$\mathbf{X}[k] = \sum_{n=0}^{N-1} \mathbf{x}[n] e^{\frac{-j2\pi nk}{N}} \tag{3.1}$$

and the inverse DFT of a sequence $\mathbf{X}$ of length N is defined as

$$\mathbf{x}[n] = \frac{1}{N} \sum_{k=0}^{N-1} \mathbf{X}[k] e^{\frac{j2\pi nk}{N}} \tag{3.2}$$

The DFT transforms one N point complex valued sequence into another. If a time domain sequence is purely real, then its DFT coefficients are symmetric:

$$\mathbf{X}[k] = \mathbf{X}[N-k]^* \tag{3.3}$$

This property can be used to compute the DFT of a real sequence almost twice as efficiently as the DFT of a complex sequence.

Block convolution methods such as overlap-add and overlap-save [14] operate on blocks of samples, and use the fast Fourier transform (FFT) algorithm to efficiently transform signals between the time and frequency domains.

Let $\mathbf{X}$ and $\mathbf{H}$ be the DFT coefficients of $\mathbf{x}$ and $\mathbf{h}$. Applying the inverse DFT to $\mathbf{Y} = \mathbf{X} \cdot \mathbf{H}$ produces the sequence $\mathbf{y}$ which is the cyclic convolution of $\mathbf{x}$ and $\mathbf{h}$. Since the length of $\mathbf{y}$ is $2N-1$ samples, it is necessary to use $2N$ point DFTs of zero-padded versions of $\mathbf{x}$ and $\mathbf{h}$ to produce $\mathbf{y}$.

If the impulse response is constant, $\mathbf{H}$ only needs to be computed once. In this case, computing a block of N output samples process requires performing one $2N$ point forward FFT, $2N$ complex multiplications, and a $2N$ point inverse FFT. An N point forward or inverse FFT takes order $O(NlogN)$ operations, so the computational cost of computing an output sample using block convolution is $O(logN)$. Beyond some impulse response length (which is system and FFT implementation dependent) block convolution is more efficient than the direct method. However, block convolution requires buffering a block of N samples before the computation can proceed, resulting in an inherent latency of $2N$ samples.

Uniform Partitioned Convolution

A compromise between latency and computational cost can be achieved by partitioning the impulse response into equally sized segments, each of which is is assigned to a separate sub-filter. The sub-filters are implemented using block convolution, and each sub-filter receives an appropriately delayed copy of the input signal. The outputs of the sub-filters are summed to produce a block of output samples. This uniform partitioned convolution (UPC) method is illustrated in Figure 3.1. If the original length-N filter is partitioned into sub-filters of size M, the computation cost per output sample is $O(NlogM)$ and the inherent latency is reduced from N to M samples.

This technique can be optimized by reusing the DFT coefficients computed by the first filter stage in later stages. A second optimization (made possible by the linearity property of the DFT) is to sum the frequency domain output spectra of the sub-filters before computing the inverse DFT. This optimization was first described by Kulp [10], and was termed a frequency-domain delay line (FDL) by Garcia [5]. After applying these optimizations, computing each block of output

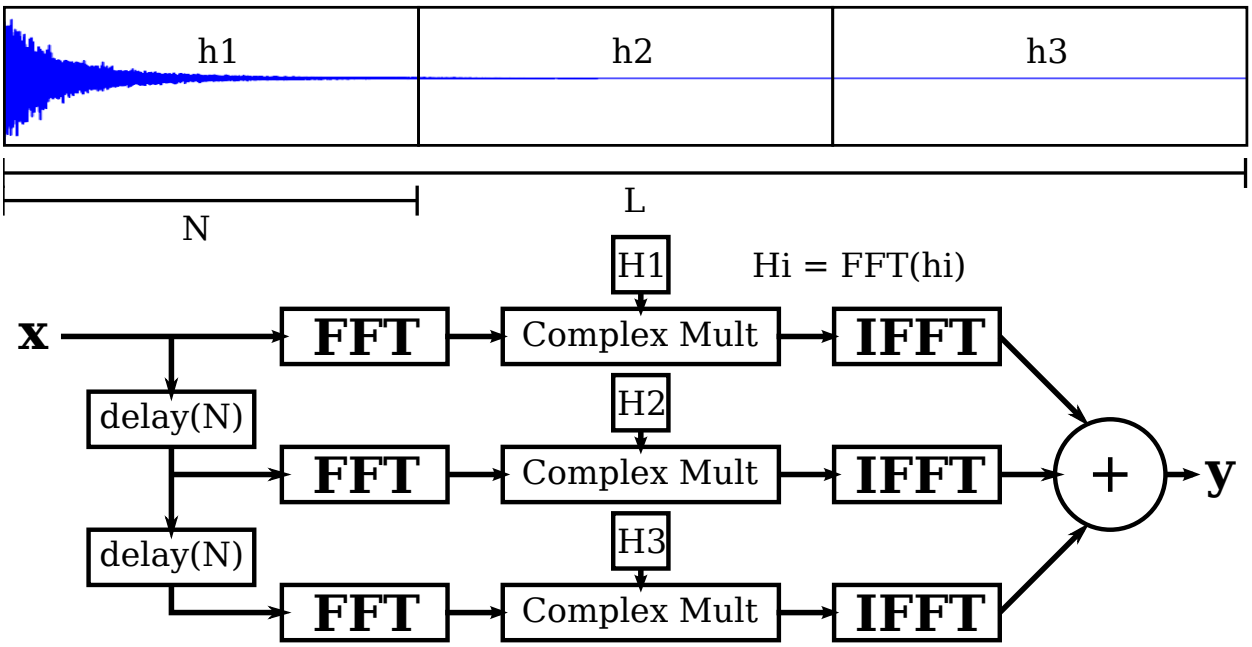

Figure 3.1: Top – Partitioning of an impulse response into 3 parts. Bottom – Steps involved in computing the above 3-part uniform partitioning.

samples requires only a single forward and inverse FFT, as shown in Figure 3.2. Even with these optimizations, the computational cost of UPC (which is dominated by the cost of the complex arithmetic performed in the frequency domain as N/M becomes large) increases linearly with the impulse response length, albeit with a smaller constant factor than the direct method.

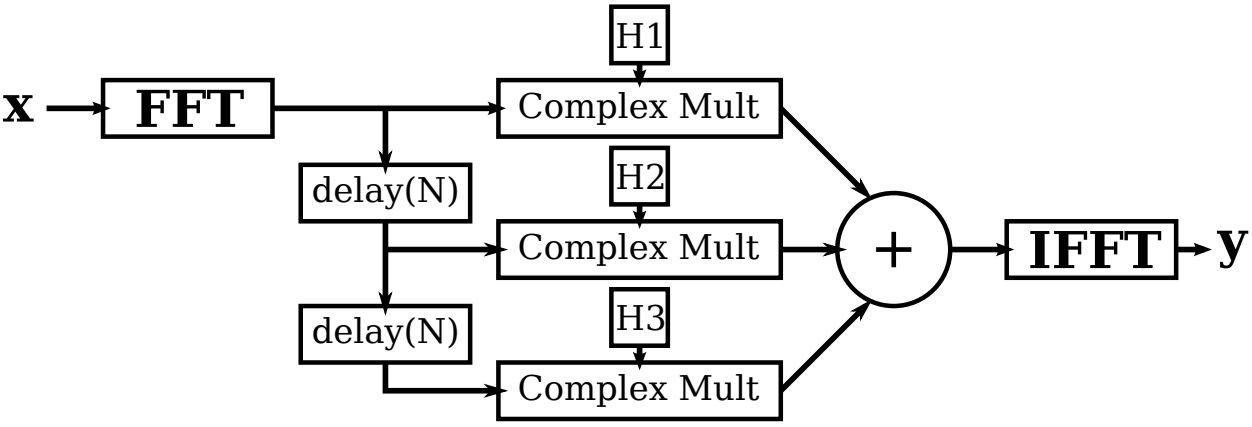

Figure 3.2: Frequency-domain Delay Line (FDL)

Non-Uniform Partitioned Convolution

Non-uniform partitioned convolution (NUPC) is a technique that improves upon the computational efficiency of the UPC method by dividing the impulse response into variable length partitions. Smaller partitions are used for the early parts of the impulse response to satisfy latency requirements, and progressively larger partitions are used for later portions of the impulse response to improve efficiency. A non-uniform partitioning can be viewed as a parallel composition of FDLs with different block sizes.

NUPC (as applied to processing audio signals in real-time) was first described by Gardner [6]. He proposes a "minimum-cost" partitioning scheme that increases the partition size as quickly as possible while maintaining a balanced processing load. This is accomplished by allowing each block convolution the same amount of time to compute a frame's worth of output as it takes to buffer a frame's worth of input. As a result, a partition of size M cannot start until at least $2M$ samples into the impulse response. For an impulse response of length $16N$, applying this scheme produces the partitioning N, N, 2N, 4N, 4N, 4N. An example of Gardner's partitioning method applied to a longer impulse response is shown at the top of Figure 3.3. Gardner's scheme tries to maximize efficiency by transitioning to larger partition sizes as quickly as possible. However his method doesn't account for the computational savings made possible by using the FDL optimization described above.

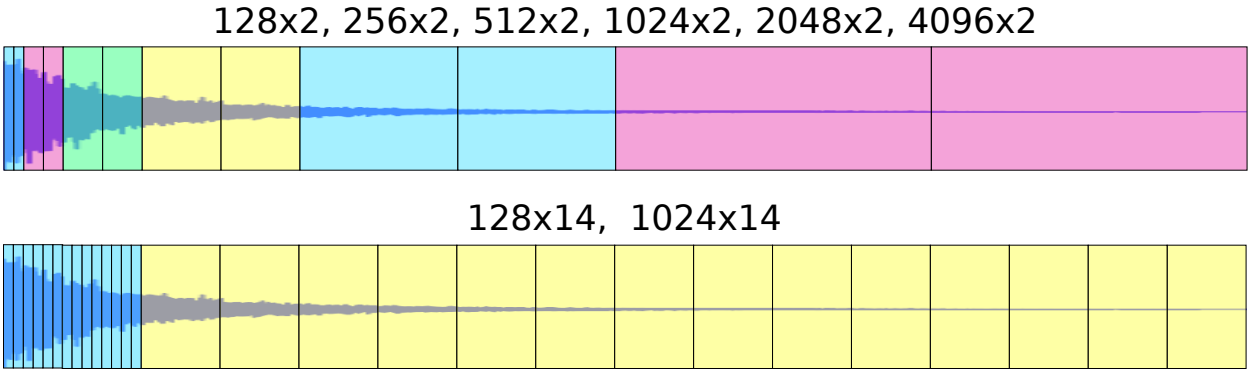

Figure 3.3: Two non-uniform partitionings of an impulse response of length 16128 with $N = 128$. Top – Gardner partitioning with 6 FDLs. Bottom – Optimal Garcia partitioning with 2 FDLs.

Another partitioning scheme was proposed by Garcia [5]. He describes a dynamic programming style algorithm that finds an optimal impulse response partitioning for a specified target

latency. His algorithm does take into account the FDL optimization, and evaluates potential partitionings using a cost function based on the number of floating point additions and multiplications required to compute an output sample. For a given impulse response length and target latency, the partitioning suggested by Garcia's algorithm is always more efficient than Gardner's "minimum-cost" partitioning. The partitioning produced by Garcia's algorithm for an example impulse response is shown at the bottom of Figure 3.3. However, since his algorithm doesn't take into account memory access latencies, the partitioning it produces may not be optimal in terms of execution time on a real machine.

We took an empirical approach to finding an optimal partitioning. Instead of picking a partitioning that is optimal with respect to some idealized cost function, we implemented an "autotuning" algorithm that measures execution times of candidate partitionings on the target machine. Since the number of possible partitionings for long impulse responses is vast, we use a dynamic programming approach to prune the search space. This enables us to find an optimal partitioning in a reasonable length of time (a few seconds). The optimal partitioning for a particular impulse response varies from machine to machine as a result of differing cache sizes and organizations, as well as other architectural variations.

Chapter 4

Implementation

We implemented the NUPC algorithm using two different scheduling paradigms: multi-threaded preemptive and single-threaded cooperative. A non-uniform partitioning is defined by the number of FDLs and the number of partitions per FDL. Each FDL is conceptually a separate task with a deadline equal to its period. The period of the primary FDL is equal to the audio I/O frame period, and the periods of larger FDLs are power of two multiples of the primary FDL period. During each audio callback period, the output of the primary FDL must be computed and summed with the outputs from the other FDLs. An FDL that is M times the size of the primary FDL must produce a block of results every M callback cycles. Figure 4.1 illustrates processing boundaries in time (arrivals and deadlines) for a partitioning with 3 FDLs.

4.1 Preemptive version

The preemptive implementation of NUPC creates a separate worker thread to perform the computation (FFTs and complex arithmetic) associated with each FDL other than the primary FDL. Since the primary FDL has a period equal to the audio I/O vector size, we perform its computation in the context of the audio I/O thread to avoid unnecessary context switches. We use the POSIX threads (pthreads) API [9] to spawn worker threads and to create condition variables (condvars) that are used for synchronization between threads. Worker threads use the SCHED_FIFO scheduling policy, with higher priorities assigned to FDLs with shorter periods.

To minimize synchronization overheads, we use system calls only when absolutely necessary.

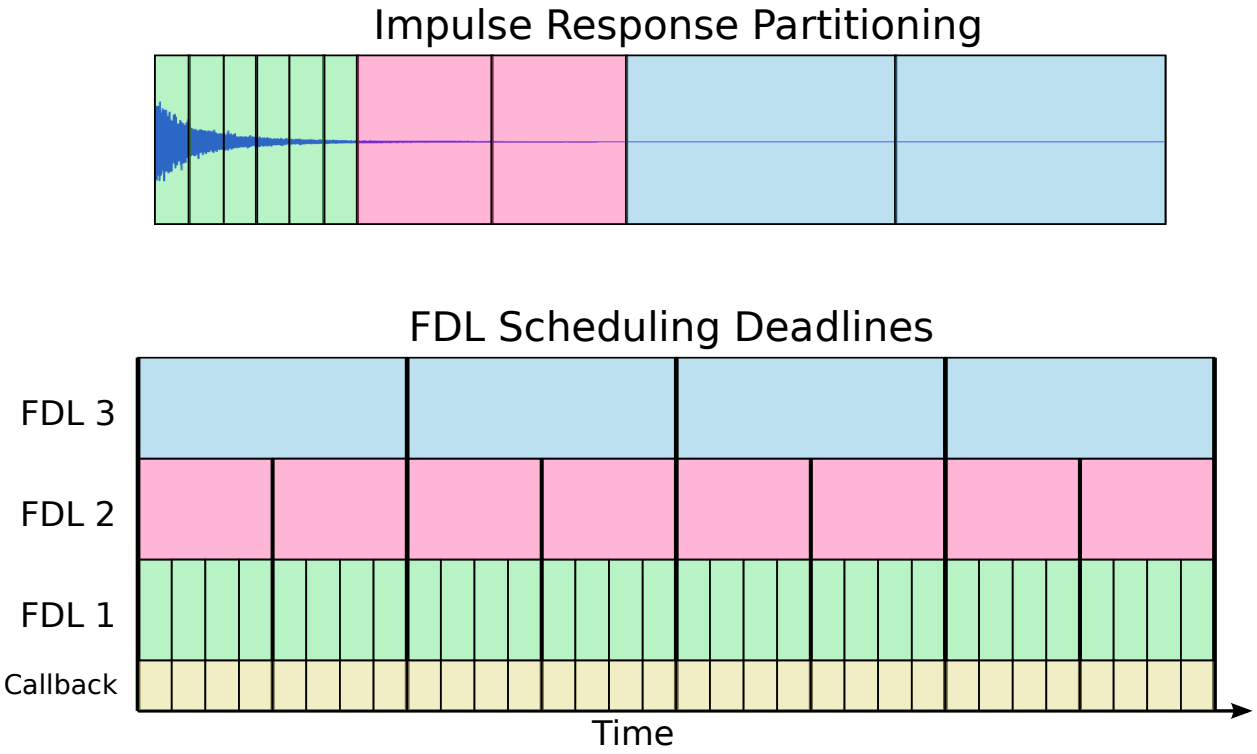

Figure 4.1: Top – Example non-uniform partitioning with 3 FDLs. Bottom – Scheduling boundaries of FDL tasks. Arrivals/deadlines are denoted by vertical lines.

When a new block of input samples is ready, the audio I/O thread wakes up waiting worker threads by performing a broadcast on a condition variable. This requires executing one system call. When a worker thread finishes computing a block of output samples, it uses an atomic memory operation (AMO) to decrement a counter and then waits on a condition variable. The last thread to finish a group of computations with the same deadline signals another condition variable to notify the audio callback thread that a new block of output is ready.

The bulk of the computation done by each worker thread is computing forward and inverse FFTs and performing complex arithmetic in the frequency domain. To do the FFT calculations, we use the FFTW[4] library. FFTW performs well on a wide range of platforms by performing an "auto-tuning" step to determine which particular set of FFT subroutines performs best on a particular system for a given FFT size. We use FFT routines that optimized to operate on real input and output sequences.

For FDLs that include many partitions, the complex multiply-add (Cmadd) routine becomes

the computational bottleneck. We wrote an optimized Cmadd routine that uses SSE3 intrinsincs to take advantage of SIMD hardware, carefully schedules instructions to minimize stall cycles, and employs aggressive loop unrolling. Our optimized Cmadd routine was roughly 8x faster than a naive version coded in C without SSE intrinsics and compiled using GCC 4.4 with aggressive optimization settings enabled.

4.2 Time-distributed version

The time-distributed implementation of NUPC performs all of its computation in the context of a single thread. During each invocation of the audio callback function, we compute the output of the primary FDL and also perform a fraction of the computation associated with all other FDLs. Since the callback function is expected to execute in a constant amount of time each time it is invoked, the distribution of work across multiple frames must be as even as possible. Implementing NUPC using this approach required significantly more programmer effort and a deeper understanding of the underlying mathematics than the previously described preemptive approach, since we weren't able to leverage external libraries (e.g. FFTW) to perform all of the computational "heavy lifting." We did, however, use FFTW to perform the "leaf-level" FFT computations. Although our time-distributed implementation isn't as flexible as the preemptive version (it only supports four FDL sizes), it conforms to the existing execution model for plugins running within host applications.

Our time-distributed implementation utilizes a technique described by Hurchalla in [7] (he calls it a "time-distributed FFT") to perform the computations required by an FDL that is 4, 8, 16 or $32\times$ the size of the primary FDL within the context of a single thread in a load balanced manner. This method takes advantage of the mathematical properties of the decimation in frequency (DIF) decomposition of a sequence.

The standard radix 2 DIF forward FFT decomposition for a DFT of size N is

$$\begin{aligned}
a[n] &= x[n] + x[n + N/2] \\
b[n] &= x[n] - x[n + N/2] \\
X[2k] &= \sum_{n=0}^{N/2-1} a[n] e^{\frac{-j2\pi nk}{N/2}} \\
X[2k+1] &= \sum_{n=0}^{N/2-1} b[n] e^{\frac{-j2\pi n}{N}} e^{\frac{-j2\pi nk}{N/2}}
\end{aligned}$$

and the corresponding inverse decomposition is

$$\begin{aligned}
a[n] &= \sum_{k=0}^{N/2-1} X[2k] e^{\frac{j2\pi nk}{N/2}} \\
b[n] &= e^{\frac{j2\pi n}{N}} \sum_{k=0}^{N/2-1} X[2k+1] e^{\frac{j2\pi nk}{N/2}} \\
x[n] &= a[n] + b[n] \\
x[n + N/2] &= a[n] - b[n]
\end{aligned}$$

A slightly modified version of the standard radix 4 DIF forward FFT decomposition for a DFT of size N is

$$\begin{aligned}
a[n] &= (x[n] + x[n + N/2]) + (x[n + N/4] + x[n + 3N/4]) \\
b[n] &= (x[n] - x[n + N/2]) - j(x[n + N/4] - x[n + 3N/4]) \\
c[n] &= (x[n] + x[n + N/2]) - (x[n + N/4] + x[n + 3N/4]) \\
d[n] &= (x[n] - x[n + N/2]) + j(x[n + N/4] - x[n + 3N/4]) \\
X[4k] &= \sum_{n=0}^{N/4-1} a[n] e^{\frac{-j2\pi nk}{N/4}} \\
X[4k+1] &= \sum_{n=0}^{N/4-1} b[n] e^{\frac{-j2\pi n}{N}} e^{\frac{-j2\pi nk}{N/4}} \\
X[4k+2] &= \sum_{n=0}^{N/4-1} c[n] e^{\frac{-j4\pi n}{N}} e^{\frac{-j2\pi nk}{N/4}} \\
X[4k-1] &= \sum_{n=0}^{N/4-1} d[n] e^{\frac{j2\pi n}{N}} e^{\frac{-j2\pi nk}{N/4}}
\end{aligned}$$

and the corresponding inverse decomposition is

$$
\begin{aligned}
a[n] &= \sum_{k=0}^{N/4-1} X[4k] e^{\frac{j2\pi nk}{N/4}} \\
b[n] &= e^{\frac{j2\pi n}{N}} \sum_{k=0}^{N/4-1} X[4k+1] e^{\frac{j2\pi nk}{N/4}} \\
c[n] &= e^{\frac{j4\pi n}{N}} \sum_{k=0}^{N/4-1} X[4k+2] e^{\frac{j2\pi nk}{N/4}} \\
d[n] &= e^{\frac{j2\pi n}{N}} \sum_{k=0}^{N/4-1} X[4k-1] e^{\frac{j2\pi nk}{N/4}} \\
x[n] &= (a[n] + c[n]) + (b[n] + d[n]) \\
x[n+N/4] &= (a[n] - c[n]) + j(b[n] - d[n]) \\
x[n+N/2] &= (a[n] + c[n]) - (b[n] + d[n]) \\
x[n+3N/4] &= (a[n] - c[n]) - j(b[n] - d[n])
\end{aligned}
$$

These decompositions have the nice property that the sub-FFTs can be computed independently of each other. Using these decompositions, we can compute an N point FFT by performing two $N/2$ point FFTs or four $N/4$ point FFTs, as well as some additions and multiplications by complex twiddle factors. Performing each sub-FFT produces a subset of the DFT coefficients of the original input sequence $\mathbf{x}$. This leads to a method for partitioning the computation (FFT, complex arithmetic, IFFT) required for block convolution across multiple time periods.

By applying a stage of radix-2 DIF decomposition to a block of input samples, the work of performing block convolution can be distributed across four frames as shown in Figure 4.2(a). In this example, the work associated with a secondary FDL that is 4$\times$ the size of the primary FDL is distributed across four processing periods.

During frames 1–4, incoming samples are buffered, and during frames 3 and 4 a radix-2 DIF decomposition is applied to the input sequence A_{in} to produce the two subsequences A_1 and A_2. The DIF decomposition can't start any earlier, since it requires the second half of the input sequence before it can produce any output. During frame 5, an FFT is performed on the subsequence A_1 and half of the resulting DFT coefficients are multiplied with those of a precomputed impulse response. The second half of the complex multiplications are performed during frame 6, after

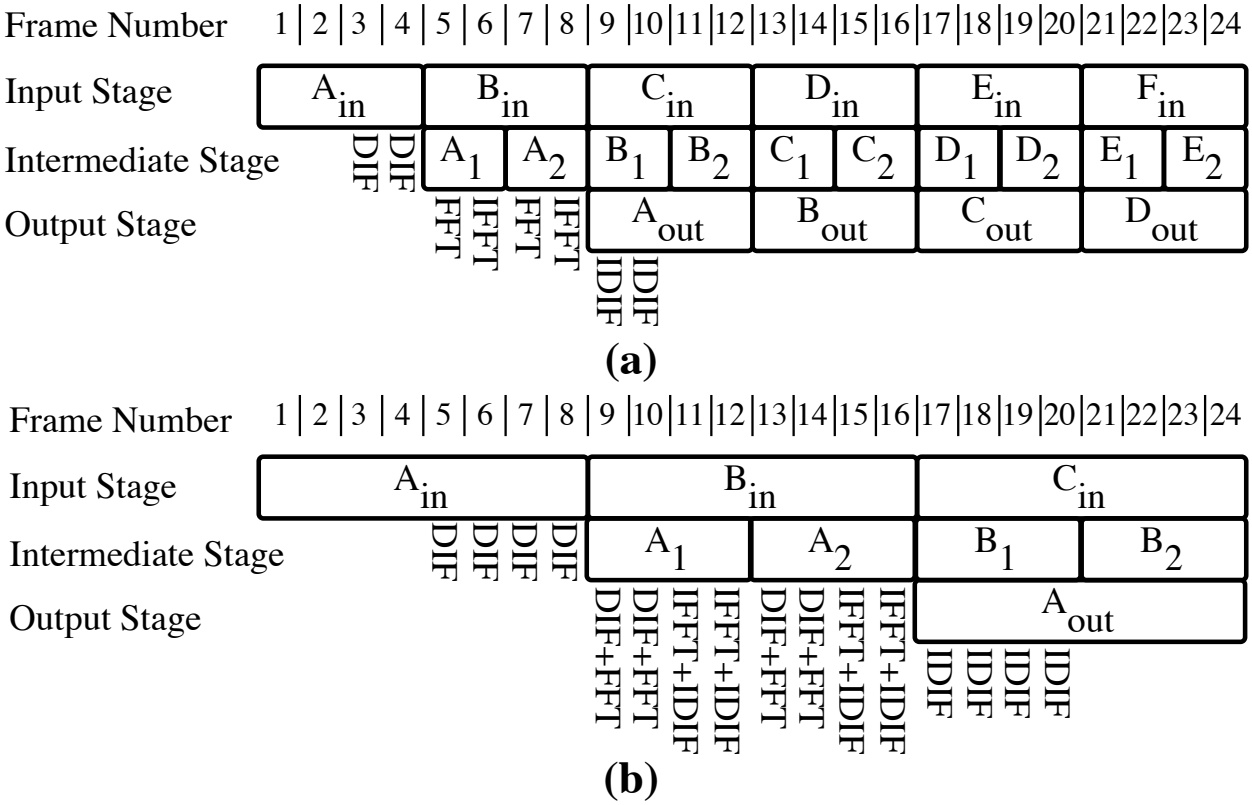

Figure 4.2: Time-distributed processing walkthrough. (a) Secondary partition 4x primary partition (b) 8x primary partition

which an IFFT is performed on the result. The same operations are performed on the subsequence A_2 during frames 7 and 8. Finally, an inverse DIF is applied to the two subsequences computed during frames 9–12 and the resulting real sequence is the output sequence A_{out}. In this example, the workload is perfectly balanced across frames since each of the 3 tasks (DIF, FFT, complex arithmetic) that are performed during each frame require the same amount of computation.

Figure 4.2(b) illustrates one way to distribute the work for a secondary FDL that is $8\times$ the size of the primary FDL by using two nested stages of radix-2 DIF. During frames 1–8, input samples are buffered and DIF is applied as in the previous example, but the work is spread out over twice as many frames (5–8). During frame 9, the first part of a radix-2 DIF is applied to subsequence A_1 to generate a new subsequence A_{11}. The FFT of this sequence is computed, and half of the complex multiplications of the resulting DFT coefficients with precomputed impulse response DFT coefficients are performed. The second part of the radix-2 DIF is computed during frame 10 to

produce the subsequence A_{12}. The FFT and half of the complex multiplications are performed, as for subsequence A_{11} during the previous frame. During frame 11, the second half of the complex multiplications started in frame 9 are completed, the IFFT of the result is computed and half of the inverse DIF is performed. Finally during frame 12 the complex multiplications started in frame 10 are finished and the second portion of the inverse DIF calculation is concluded, resulting in a complete output sequence corresponding to the input sequence A_1. The same computations are performed on the sequence A_2 during frames 13–16, and the resulting output sequence is combined with the one from frames 9–12 (using inverse DIF) to produce the output sequence A_{out} during frames 17–20. Unlike the previous example, in this case the workload isn't perfectly balanced because the work associated with the nested forward and inverse DIF steps varies somewhat across frames.

To partition the work for an FDL that is 16$\times$ the size of the primary FDL across 16 frames, we can use a radix-4 DIF in place of the radix-2 DIF used during the nested decomposition step (periods 9–16 in the last example). A radix-4 DIF decomposition produces four subsequences instead of two, enabling us to distribute the work over twice as many processing periods. If we use two nested stages of radix-4 DIF decomposition, we can partition the work of an FDL that is 32$\times$ the size of the primary FDL across 32 frames. However, using higher radix decompositions results in greater variation of the processing load across frames.

We use a technique described by Vernet [21] for efficiently performing block convolution on two real sequences using half length complex FFTs. Two N point real sequences x and h are packed into the real and imaginary parts of two $N/2$ point complex sequences and then multiplied by a set of complex twiddle factors to produce a and b:

$$a[n] = (x[n] + jx[n + N/2]) \cdot e^{\frac{-j2\pi n}{2N}}$$
$$b[n] = (h[n] + jh[n + N/2]) \cdot e^{\frac{-j2\pi n}{2N}}$$

$N/2$ point complex FFTs are performed on a and b to generate A and B, which are multiplied to produce C. Performing an inverse FFT on C gives us c, which is multiplied by twiddle factors to

produce d, from which the desired real output sequence y can be recovered as follows:

$$d[n] = c[n] \cdot e^{\frac{j2\pi n}{2N}}$$
$$y[n] = Re\{d[n]\}$$
$$y[n + N/2] = -Im\{d[n]\}$$

Applying this technique to the algorithms described above allows us to use complex FFT routines exclusively without sacrificing efficiency. Otherwise, we would need to use of a mix of FFT routines optimized for real and complex sequences. Using the same FFT routines throughout the algorithm results in greater symmetry and better load balancing.

In [8], Hurchalla describes a method for applying nested short-length acyclic convolution algorithms to improve the computational efficiency of the complex arithmetic performed in the frequency domain. The basic idea is to treat each frequency bin in each partition of the impulse response as a sequence, and to perform a running convolution between this sequence and the corresponding frequency bin of the FFT of the input signal. We implemented a basic version of Hurchalla's scheme, using a single stage of 3-partition acyclic convolution. These convolution routines, as well as the routines used to perform the forward and inverse radix-2 and radix-4 decomposition steps, were hand optimized in assembly using the SSE3 extensions to the x86 ISA. While this scheme did reduce the overall amount of work done (in terms of the total number of floating point operations executed), we found that the variation in execution time from frame to frame was greater than when using a naive implementation of convolution. This resulted in a longer worse case execution time, which meant that the version of the code that used the optimized convolution routines was never able to concurrently process as many independent channels of convolution as the version using the naive convolution routines. For this reason, we do not include an evaluation of the code using this optimization in the evaluation section. Hurchalla also discusses various techniques to time distribute work across multiple frames when working with multiple channels. We did not implement any of these techniques – when operating with multiple channels, our implementation processes each channel independently.

Chapter 5

Evaluation

In this section we present and analyze performance measurements of our preemptive and time-distributed implementations of non-uniform partitioned convolution. The machine used in these experiments was a Mac Pro with dual 2.66 GHz 6-core Intel Xeon "Westmere" processors and 12GB of memory, running Linux kernel version 3.1.0-rc4 with the CONFIG_PREEMPT kernel configuration option enabled. This option reduces interrupt and scheduling latencies by making all kernel code that is not executing in a critical section preemptible. We only enabled one of the two sockets in the system and disabled Hyperthreading during all of the experiments. The built-in audio device was used for I/O, set to a sample rate of 44.1 kHz and an I/O vector length of 32 or 64 samples. Both versions interface with the audio subsystem using the ALSA API directly – as opposed to using a cross-platform library (such as PortAudio) or daemon (such as JACK) – to minimize latency and overhead. We ran each experiment using impulse responses that ranged in length between 16,384–524,288 samples (0.4–11.9 seconds).

Both versions were implemented as standalone applications that take arguments which specify how many channels (instances) of convolution to perform and the impulse responses and partitionings to use. The time-distributed implementation used the same partitioning for all the experiments: two FDLs, with the secondary FDL 32$\times$ as large as the primary FDL. We chose to use this partitioning in our evaluation because it most clearly demonstrates the consequences of the uneven load balancing across frames in the time-distributed implementation. The preemptive implementation uses different partitionings for different experiments, and spawns a single thread per FDL that it uses to process all channels (instead of spawning a separate thread per channel per FDL) to avoid

unnecessary context switches.

To make our results as deterministic as possible, we disabled all frequency scaling mechanisms present in the operating system, as well as Turbo Boost (hardware based opportunistic frequency scaling) in the CPU. We killed all non-essential processes (including X11) and used SSH to log in to the machine through an ethernet port to start the experiments.

One of the main reasons we chose Linux as our evaluation platform was our inability to get acceptably deterministic results when performing experiments under MacOS. MacOS provides no way to disable frequency scaling or Turbo Boost, nor does it provide mechanisms to pin threads to cores. Another factor that contributed to our decision was the complex and relatively opaque nature of the MacOS audio framework (CoreAudio) and the OS in general. While ALSA and the Linux kernel are not terribly well documented, having access to the source code makes it possible to understand the observed behaviors of the system (given enough time and determination!).

5.1 Single-Core Performance

Our first experiment was to record the OS reported CPU utilization for three different configurations running the same workload on a single core. We disabled every core in the system but one, so all interrupt handlers and other OS tasks were running on the same core as the convolution app. The reported number reflects the total CPU utilization for the core (not just for the app) and is averaged over a three second interval. The workload was 16 independent channels of NUPC, and the three configurations were: preemptive using two FDLs (PE2), preemptive using the empirically derived optimal partitioning with between 3–5 FDLs (PEO), and time-distributed (TD). The CPU utilization values are presented in Figure 5.1, and Figure 5.2 shows the execution times of individual threads for the PE and PEO configurations.

TD and PE2 both use the same partitioning so we would expect them to exhibit a similar computational load, and they do. PEO uses a partitioning with three or more FDLs, so it actually performs sigificantly less work per output sample than either PE2 or TD, and explains why it outperforms PE2 and PEO by a wide margin. The reported CPU utilization for TD and PE2 are within a few percent, with PE2 having a slight advantage despite the overhead associated with preemption.

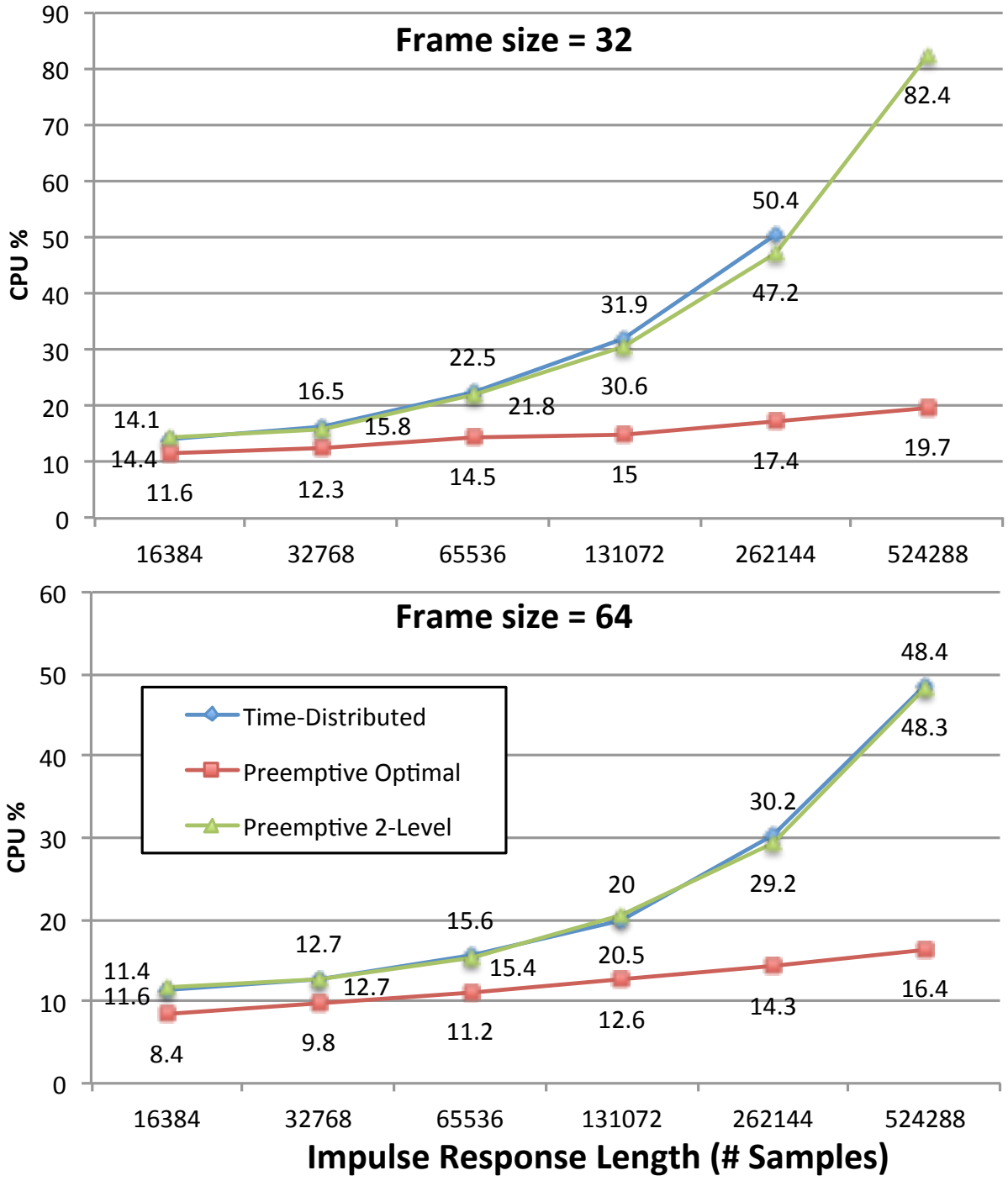

Figure 5.1: CPU utilization of a single core while performing 16 channels of convolution.

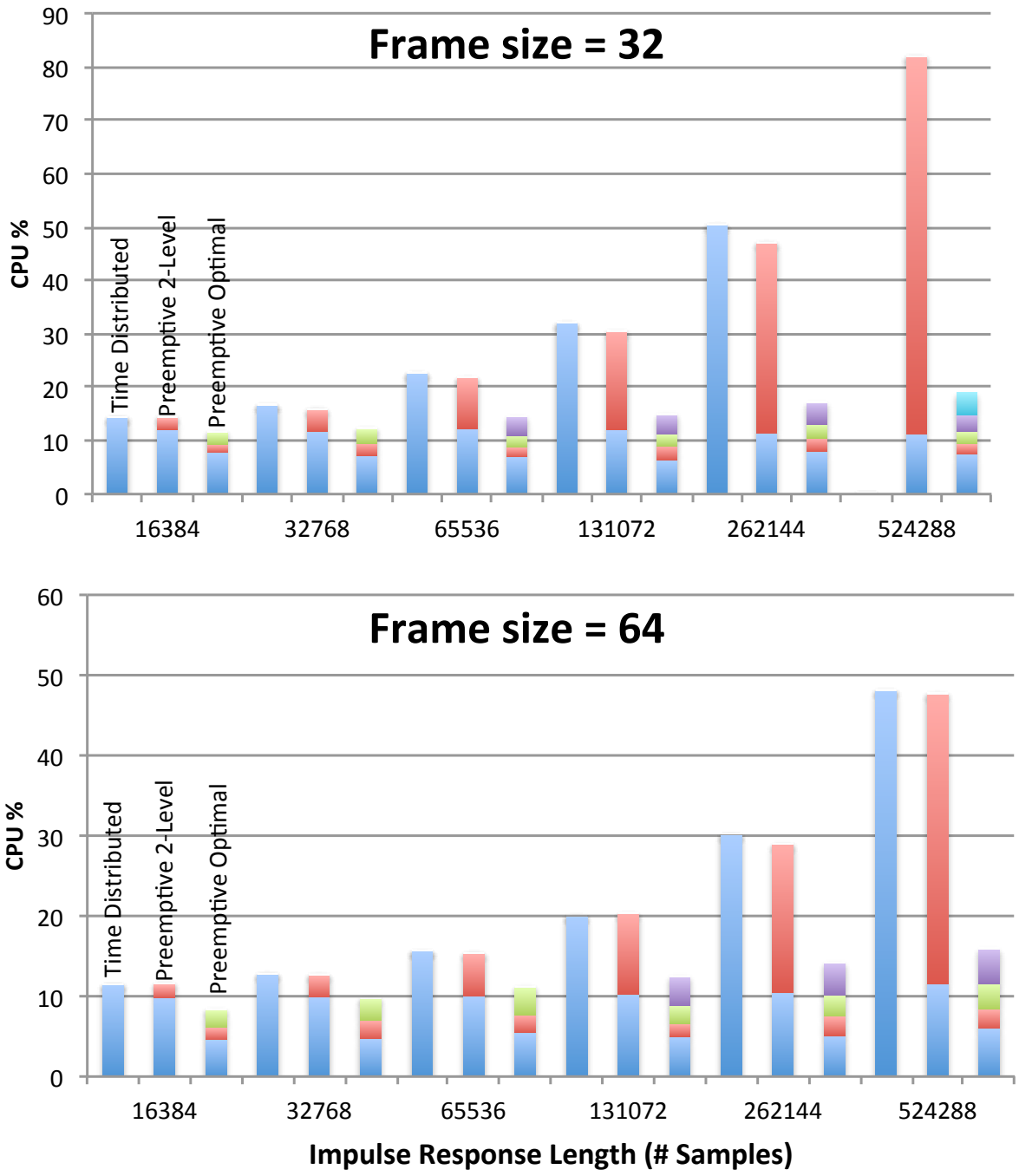

Figure 5.2: Breakdown of CPU utilization per thread

PE2 has memory access patterns with better spatial and temporal locality and predictability than TD, and we believe these are responsible for giving it a slight edge in performance relative to TD. During each invocation of the audio callback function, TD performs $1/32$nd of the total work of computing one channel's secondary FDL, then does the same for the next channel, and so on. In contrast, the worker thread in PE2 computes the output of each channel's secondary FDL to completion before moving on to the next channel (though it may be preempted). This means that PE2 generates long streams of unit stride memory accesses while executing the Cmuladd loop, which enable it to benefit from the hardware prefetch engines in the memory hierarchy. This results in fewer stall cycles waiting for cache misses to resolve which improves performance. Additionally, TD performs two (one per FDL) $2N$ (where N is the audio I/O vector length) point forward and inverse FFTs for each channel during each audio callback, whereas PE2 performs one $2N$ forward and inverse FFT per channel during each audio callback, and one $64N$ point forward and inverse FFT per channel in the worker thread over the course of 32 callback periods. The larger FFTs performed by PE2 benefit from improved spatial and temporal locality in their memory access patterns as compared to the many smaller FFTs (spread out over time) performed by TD.

Our second experiment was to measure how many independent channels of convolution each configuration could sustain without missing any deadlines and causing dropouts in the output audio stream. This experiment was also performed using a single core. We progressively increased the channel count until we reached the highest value that ran dropout–free for 60 seconds. The results are presented in Figure 5.3. As in the previous experiment, the improved computational efficiency of PEO means it outperforms TD and PE2. However, PE2 is able to process significantly more channels without dropouts than TD. This is somewhat surprising, given that TD and PE2 had almost identical CPU utilization as measured in the previous experiment. We believe this is due to two factors: the imperfect load balancing of TD and the reduced sensitivity to variations in the scheduling latency of the callback thread of PE2.

The previous experiment measured reported CPU utilization averaged across thousands of frames, but the worst-case execution time (WCET) during any individual frame determines whether or not a deadline is met. In the TD implementation, the load balancing is not perfect and so the WCET is higher than the average execution time. During the course of these experiments, the largest difference we measured between the worst case and average execution time across

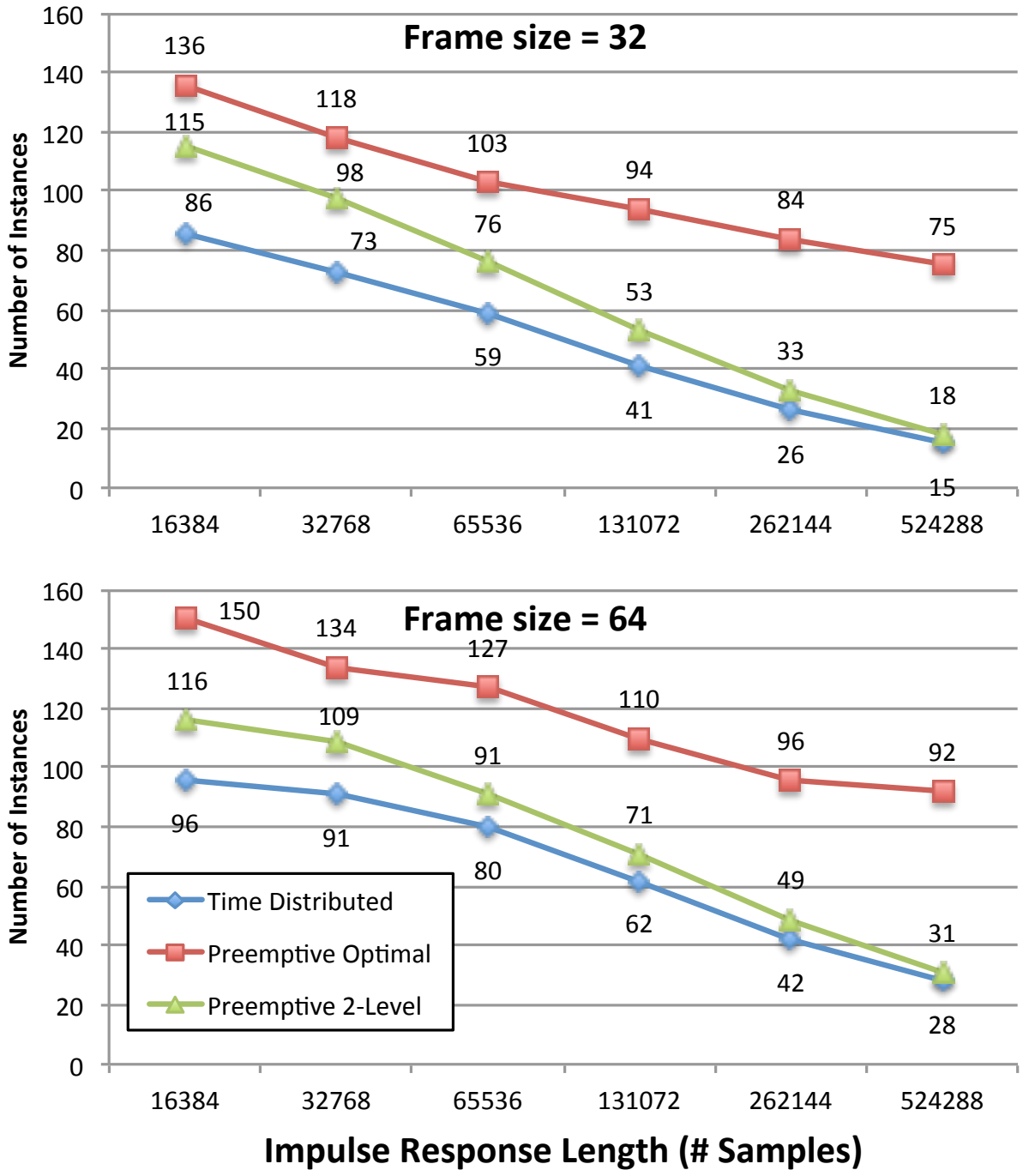

Figure 5.3: Maximum number of independent channels of convolution possible without dropouts for three configurations running on a single core.

frames for TD was approximately 15%, though it varied significantly for different impulse response lengths and FDL sizes.

TD is more sensitive to variations in the scheduling latency of the callback thread than PE2. If the callback function starts late, that means there is less time available for processing without missing a deadline. TD does all of its work in the context of the audio callback thread, whereas PE2 does part of its work in the callback thread and part of it in the worker thread. Since PE2 has to spend less time in the callback thread before a block of output is ready, it is less likely to experience a missed deadline if a callback starts later than usual. PE2's worker thread only needs to produce a result every 32 frames and is relatively unaffected by variations in the scheduling latency of the callback thread.

5.2 Multi-Core Performance

Our final experiment was to measure how many channels of convolution the preemptive implementation could sustain when running on multiple cores. Each worker thread (one per FDL) was pinned to a separate core to minimize OS scheduler induced non-determinism, and any cores that weren't necessary for a given experiment were disabled. Because the number of FDLs in a partitioning varies with the impulse response length, so do the numbers of cores used in these experiments.

We also considered an alternative scheme where channels (instead of FDLs) were partitioned across cores. In this scheme, one thread per FDL level was pinned to each core – so for N FDLs and M cores there would be a total of $N \times M$ threads active in the system. However, the pin-by-FDL scheme outperformed the pin-by-channel scheme in all of our measurements, so we only present the results from the former here.

The performance of the single and multi-core versions of PE is presented in Figure 5.4. By using additional cores, we were able to process between 1.23 – 1.83$\times$ as many channels without experiencing dropouts. While our work partitioning scheme is probably not optimal (since there is significant variation in the computational load across FDLs), we believe the factor that ultimately limits the maximum achievable number of independent instances is memory bandwidth, not computational crunch. The CPU used in these experiments has 12MB of shared last level cache and 256KB of private level 2 cache per core. A 512k point impulse response represented as single precision floating point values occupies 2MB of memory. When processing many channels with long

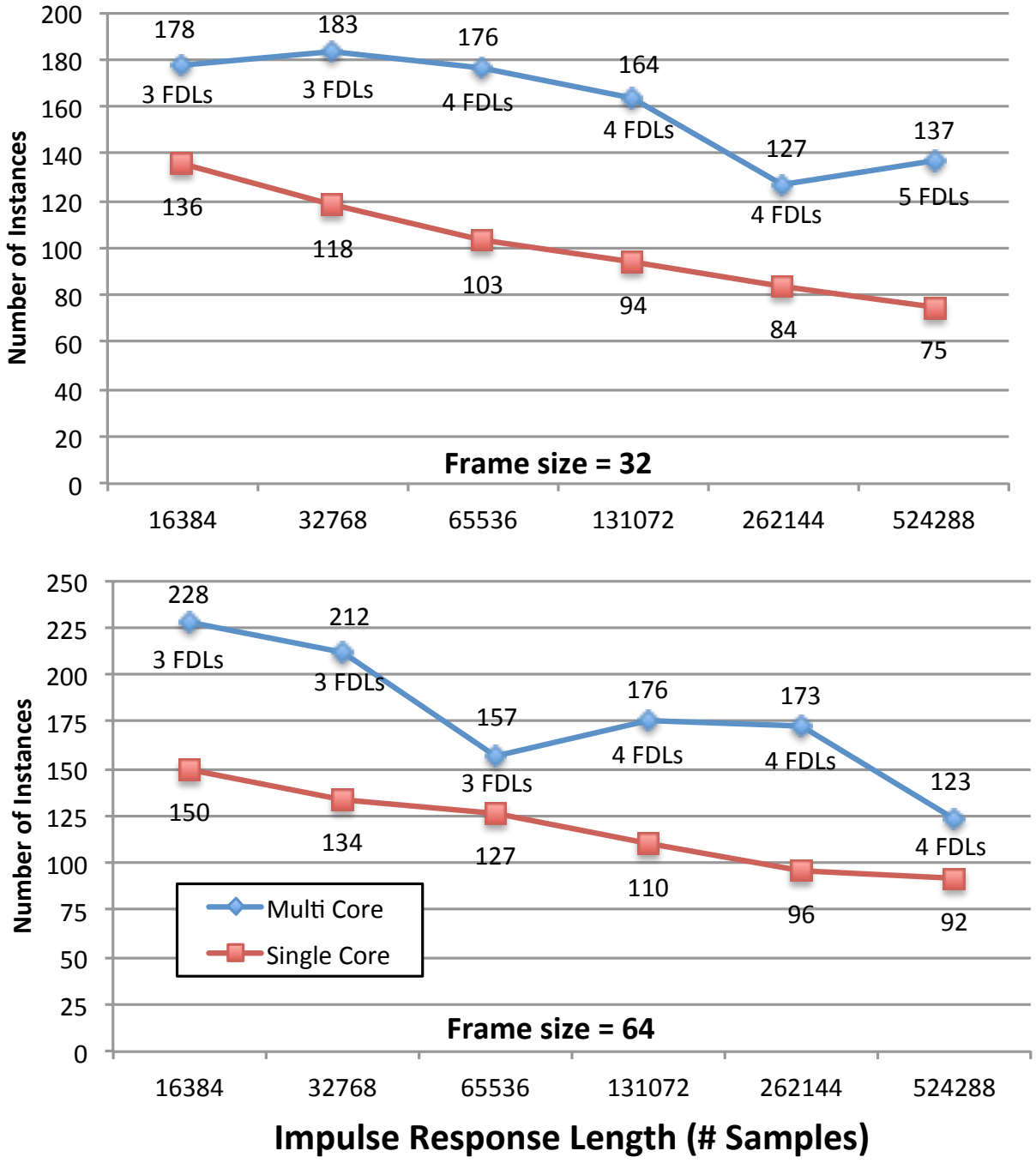

Figure 5.4: Maximum number of independent channels of convolution possible without dropouts for single and multi-core cases (preemptive implementation using optimal partitioning). Points are labeled with the number of FDLs used.

impulse responses, the working set doesn't fit into the on-chip caches and the latency of DRAM accesses impacts performance.

Chapter 6

Discussion

In all of the experiments we conducted, the preemptive version of NUPC (using an empirically determined optimal partitioning) outperformed the time-distributed version by a wide margin. Our motivation for implementing the time-distributed version was to enable us to write a plugin that would behave deterministically when executing in the context of current audio host applications.

The preemptive version required significantly less programmer effort to implement than the time-distributed version. While efficiently managing the scheduling and synchronization of multiple threads is not trivial, it allows us to use existing highly optimized libraries to implement processing tasks without concerning ourselves with partitioning and load balancing. Optimizing the time-distributed FFT to the point that its performance was competitive with FFTW's FFT routines required hand tuning assembly code and carefully managing the layout of data structures in memory. This was a time-consuming process and required intimate knowledge of the intricacies of the CPU microarchitecture and memory hierarchy. Requiring plugin implementors to perform these sorts of optimizations to achieve reasonable performance for multi-rate algorithms seems like an unreasonable burden. Furthermore, the techniques used to implement the time-distributed FFT don't scale well to larger (greater than $32\times$) FDL sizes, limiting the performance of the time-distributed NUPC implementation for very long impulse response lengths. There do exist opportunities to further optimize the time-distributed implementation – though the obvious improvements, such as taking advantage of the regularity across channels to more optimally distribute the computation [7] would require additional programmer effort and would only benefit specific use cases.

Partitioned convolution is just one example of a class of multi-rate audio processing and analysis tasks – others include score following, rhythm and pitch extraction, and algorithmic composition. Generally speaking, it can be quite cumbersome (if not impossible) for a programmer to time-distribute long-running tasks evenly across multiple short time periods, particularly if they would like to take of advantage existing external libraries. In the case of FFTs, there are clever mathematical manipulations that enabled us to accomplish this in a limited manner but other algorithms (for example, those related to machine learning) may not be as amenable to such treatment.

While it is possible for plugins to spawn OS threads while running in the context of existing audio host applications, there is no guarantee that other plugins running on the host won't do the same thing. This would result in pollution of the "thread ecosystem," forcing threads with real-time constraints to contend with each other for access to processing resources. Ultimately, when there are more active threads than cores in a system (oversubscription), the burden of making scheduling decisions falls onto the OS. Given that the OS has very limited knowledge about the relationships and dependencies between threads in an application, it is unreasonable to expect it to make optimal scheduling decisions.

In the future, we believe that operating systems and plugin APIs must be extended with mechanisms to enable audio host applications to schedule the execution of their processing tasks across multiple cores while also supporting preemption. This is necessary to enable developers to efficiently write audio plugins and applications that can take advantage of the compute resources in current and future multi-core CPUs. These mechanisms should satisfy three objectives: efficient and deterministic scheduling of tasks within an application, hierarchical composition of schedulers, and support for preemption. To this end, we propose the adoption of a two-level scheduling model. In such a model, the OS gang schedules an application's threads and a user-level scheduler (ULS) within the application handles the mapping of processing tasks onto processor cores. While such two-level scheduling ecosystems are not a new concept, none of the existing proposals we are aware of support both preemption and the hierarchical composition of user-level schedulers.

The basic idea behind user-level threads and schedulers is to time-multiplex many user-level threads (or tasks) onto fewer kernel threads. Kernel threads used in this way are sometimes called "virtual processors." Context switching and synchronization between user-level threads is efficient since it can be done without any OS involvement. Cappriccio [3] is an example of a user-level thread package that implements thread management and synchronization. Cappriccio assumes a

purely cooperative scheduling paradigm and requires threads to perform I/O using non-blocking system calls and event-based notifications. This illustrates a major shortcoming of implementing user-level threading on conventional operating systems: if a user-level thread executes a blocking system call, its associated kernel thread will block and the ULS will lose control of that thread until the kernel thread is unblocked. Preemption can also be problematic, since if a user-level thread is preempted while holding a lock, other threads may be prevented from making forward progress. In both situations, there's no way for the OS to notify a ULS that one of its threads has blocked or been preempted.

Several OS mechanisms have been proposed to address this issue. Scheduler activations [1] transfer control from the kernel to a user-level thread scheduler to notify it about kernel events such as blocked threads, and to provide a context in which begin executing another thread. First class user-level threads [12] use "software interrupts" to notify an application about kernel events related to its threads. In both schemes, notifying user-space about a kernel event is a fairly expensive operation that may involve multiple kernel crossings, making them not well suited for use in real-time contexts. Issues related to using user-level threads in real-time contexts are discussed in [20] and [15]. None of these proposals address the issue of hierarchically composing multiple ULS within an application, nor have any of them been implemented on widely available operating systems for general pupose computers.

Lithe [18] is a low-level substrate which provides mechanisms and a standard API to enable multiple parallel libraries to be composed within an application efficiently. Lithe provides a solution to the composability problem, but currently doesn't support preemption or deal with blocking system calls. We believe that extending Lithe beyond to support preemption (which will likely require implementing new OS mechanisms) and requiring that plugins use Lithe's mechanisms to implement tasks that would benefit from parallel processing is the best way to address the issues we have identified. The best way to accomplish this goal is an open research problem, which we plan to address in future work.

Bibliography

[1] Thomas E Anderson et al. “Scheduler activations: effective kernel support for the user-level management of parallelism”. In: *Transactions on Computer Systems (TOCS)* 10.1 (Feb. 1992).

[2] Rimas Avizienis and Eric Battenberg. “Implementing Real-Time Partitioned Convolution Algorithms on Conventional Operating Systems”. In: *Proc. of the 14th Int’l Conference on Digital Audio Effects* (2011).

[3] Rob von Behren et al. “Capriccio: scalable threads for internet services”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 268–281.

[4] Matteo Frigo and Steven G. Johnson. “The Design and Implementation of FFTW3”. In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231.

[5] G Garcia. “Optimal filter partition for efficient convolution with short input/output delay”. In: *Audio Engineering Society Convention 113* (2002).

[6] WG Gardner. “Efficient convolution without input-output delay”. In: *Journal of the Audio Engineering Society* (1995).

[7] J Hurchalla. “A time distributed FFT for efficient low latency convolution”. In: *Audio Engineering Society Convention 129* (2010).

[8] J Hurchalla. “Low Latency Convolution in One Dimension Via Two Dimensional Convolutions: An Intuitive Approach”. In: *Audio Engineering Society Convention 125* (2008).

[9] IEEE/ISO/IEC. *ISO-IEC 9948-1: IEEE Std. 1003.1-1996 Information Technology Portable Operating System Interface (POSIX) Part 1 System Application Program Interface (API) [C Language]*. New York, NY, USA: IEEE Standards Office, 1996. ISBN: 1559375730.

[10] B.D. Kulp. “Digital equalization using fourier transform techniques”. In: *In Proceeding of 85th AES Convention, Los Angeles* (1988).

[11] S. Letz, Y. Orlarey, and D. Fober. “Jack audio server for multi-processor machines”. In: *Proceedings of the International Computer Music Conference*. 2005.

[12] Brian D. Marsh et al. “First-class user-level threads”. In: *ACM SIGOPS Operating Systems Review* 25.5 (Sept. 1991), pp. 110–121.

[13] Robert A. Moog. “MIDI: Musical Instrument Digital Interface”. In: *J. Audio Eng. Soc* 34.5 (1986), pp. 394–404.

[14] H.J. Nussbaumer. *Fast Fourier transform and convolution algorithms*. Fast Fourier Transform and Convolution Algorithms. Springer-Verlag, 1982. ISBN: 9780387118253.

[15] Shuichi Oikawa and Hideyuki Tokuda. “Reflection of developing user-level real-time thread packages”. In: *ACM SIGOPS Operating Systems Review* 29.4 (1995), pp. 63–76.

[16] Alan V. Oppenheim and R.W. Schafer. *Digital signal processing*. Prentice-Hall, 1975. ISBN: 9780132146357.

[17] Davis Pan. “A Tutorial on MPEG/Audio Compression”. In: *IEEE MultiMedia* 2 (1995), pp. 60–74.

[18] Heidi Pan, Benjamin Hindman, and Krste Asanović. “Composing parallel software efficiently with lithe”. In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI ’10. New York, NY, USA: ACM, 2010.

[19] Francis Rumsey. “Digital Audio Interfacing-A Brief Overview (Digital Audio Tutorial)”. In: *Audio Engineering Society Conference: 10th International Conference: Images of Audio*. Sept. 1991.

[20] Yangmin Seo et al. “Supporting preemptive multithreading in the ARX real-time operating system”. In: *TENCON 99. Proceedings of the IEEE Region 10 Conference*. 1999, pp. 443–446.

[21] J L Vernet. “Real signals fast Fourier transform: Storage capacity and step number reduction by means of an odd discrete Fourier transform”. In: *Proceedings of the IEEE* 59.10 (), pp. 1531–1532.

[22] Matthew Wright, Adrian Freed, and Ali Momeni. “OpenSound Control: state of the art 2003”. In: *Proceedings of the 2003 conference on New interfaces for musical expression*. NIME ’03. 2003.