

IMPLEMENTING REAL-TIME PARTITIONED CONVOLUTION ALGORITHMS ON CONVENTIONAL OPERATING SYSTEMS

Eric Battenberg, Rimas Avizienis

Department of Electrical Engineering and Computer Science
University of California, Berkeley, CA
{erichb, rimas}@eecs.berkeley.edu

ABSTRACT

We examine real-world techniques for implementing partitioned convolution on conventional operating systems. We discuss the optimization of two different scheduling paradigms, preemptive and time-distributed, and evaluate their performance when used to process large numbers of audio channels using varying impulse response sizes at low throughput latency. We find that while the time-distributed version fits better within the context of current audio host programs, the preemptive version was much easier to implement, operates at lower cpu load, and behaves more reliably, despite the fact that it relies on preemptive context switching between multiple threads.

1. INTRODUCTION

Partitioned convolution is a technique for efficiently performing time domain convolution with low inherent latency[1]. It is particularly useful for computing convolutions of audio signals with long impulse responses (> 1 second) in real time, as direct convolution becomes too computationally expensive and block FFT convolution incurs unacceptable latency.

Much of the existing work on partitioned convolution has focused on optimizing the computational pieces of the algorithm, i.e. efficiently implementing FFTs and spectral multiplications[2][3][4] and finding computationally optimal partitionings[?]. In this paper, we focus on how to most effectively schedule the necessary computations on a personal computer using a conventional operating system.

We investigate the performance of two scheduling approaches for non-uniform partitioned convolution: a preemptive, multi-threaded approach and a cooperative, time-distributed approach. Important areas of investigation include performance, reliability, compatibility with existing audio hosts, and programming difficulty.

In Section 2, we cover the basics of non-uniform partitioned convolution. Sections 3 and 4 cover the implementation of the preemptive and cooperative approaches, respectively, and Sections 5 and 6 present and discuss the performance results and implications of the two scheduling approaches.

2. ALGORITHM OVERVIEW

Convolution is a mathematical operation typically used to perform finite impulse response (FIR) filtering on a signal:

$$y[n] = \sum_{k=0}^{L-1} x[k]h[n-k] \quad (1)$$

Where x and y are the input and output signals, respectively, and h is the length- L impulse response of the FIR filter.

The above direct method of convolving two signals has no inherent latency but carries with it a large computational cost per output sample ($O(L)$ multiply-adds per output sample). Because of this, real-time convolution with larger impulse responses is usually carried out using block FFT-based methods, like overlap-add and overlap-save, which take the FFTs of the impulse response and a buffered portion of the input signal, multiply them together in the frequency domain, and take the inverse FFT of their complex product to compute a portion of the output signal [5]. Computing convolution in this way requires significantly less computation ($O(\log L)$) at the expense of increased latency due to buffering.

2.1. Uniform Partitioned Convolution

In order to obtain a compromise between computational efficiency and latency, we can partition the impulse response into a series of smaller sub-filters which can be run in parallel with appropriate delays inserted. Each sub-filter's output is computed using a block-FFT method, and the outputs of all sub-filters are summed to produce a block of the output signal, as shown in Figure 1.

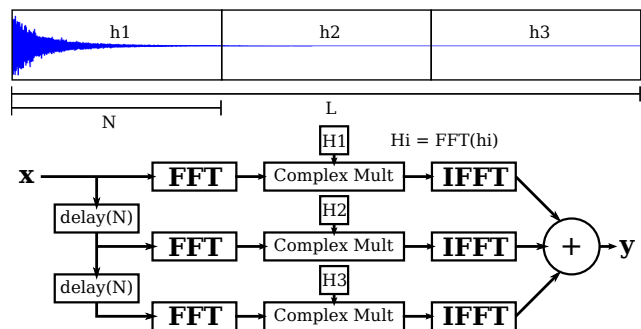


Figure 1: Top - Partitioning of an impulse response into 3 parts. Bottom - Steps involved in computing the above 3-part uniform partitioning.

If the original length- L filter is partitioned into sub-filters of size N , we perform $O(\log N)$ operations per sub-filter per output sample but have reduced the latency from L to N . This scheme works well but can become infeasible if low latency is required for filters longer than a second or two. For example, if 1.5ms processing latency is desired (64 samples at 44.1kHz), a 92ms (4096 sample) filter would need to be cut into 64 sub-filters, while a 6sec (262144 sample) filter would require 4096 sub-filters.

Within this uniform partitioning scheme, we can save previously computed forward FFTs for reuse in subsequent sub-filters. Additionally, linearity of the FFT allows us to sum the complex frequency domain output of each sub-filter before taking the inverse FFT, which reduces the number of inverse FFTs required to one. In Figure 1, these optimizations would be made by replacing the FFT/IFFTs with a single FFT before the delays and a single IFFT after the sum as shown in Figure 2. Because the delays now take place in the frequency domain, Garcia refers to this method of computing a uniform partitioning as a *Frequency-domain Delay Line (FDL)* [?]

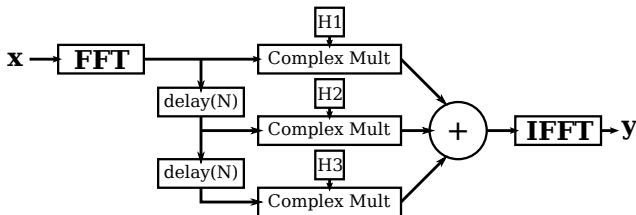


Figure 2: *Frequency-domain Delay Line (FDL)*

Even with this FDL optimization, the computational cost of uniform partitioned convolution (primarily the number of complex multiplications and additions of frequency domain coefficients) scales linearly with the impulse response length and its use becomes impractical for very long impulse responses.

2.2. Non-Uniform Partitioned Convolution

Non-uniform partitioned convolution attempts to improve upon the computational efficiency of the uniform partitioned convolution method by dividing the impulse response into partitions of various sizes. The approach is to use shorter partitions near the beginning of the impulse response to achieve low latency and longer partitions towards the end to take advantage of increased computational efficiency. In [1], Gardner describes how to use such a mix of partitions sizes to improve efficiency without sacrificing latency.

Gardner suggests a partitioning scheme that increases the partition size as quickly as possible, as shown at the top of Figure 3. However, Garcia points out that since the FDL optimization can be applied to each block size used in a non-uniform partitioning, it is usually more efficient to use more FDLs of a given size (bottom Figure 3) before moving to a larger partition size [1]. Therefore, we can view a non-uniform partitioning as a parallel composition of FDLs of increasing block size.

Longer FDLs can be further optimized using Hurchalla's *Nested Acyclic Convolution* technique that allows many same-size convolutions to be done at a reduced computational cost [4]. It is also possible to reduce FFT related computations by computing the larger FFTs from the intermediate values of smaller FFTs [1, 6]. Additional computational improvements include enhancing FFT and complex arithmetic efficiency via system-specific optimizations such as using SIMD instructions and cache-aware tuning [7].

FDL-based non-uniform partitioned convolution is a very computationally efficient approach to low-latency real-time convolution; however, a real-world implementation of this approach still leaves many decisions to be made, the most important being: how should we schedule all of this computation on a conventional operating system? We cover two approaches to scheduling in the following two sections.

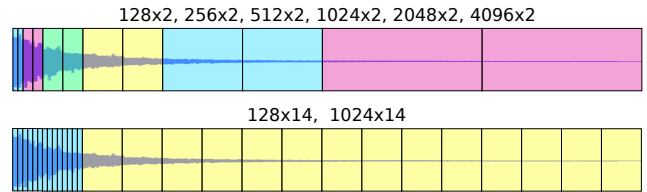


Figure 3: *Two non-uniform partitionings of a length 16128 impulse response with $N = 128$. Top - Gardner partitioning with 6 FDLs. Bottom - Optimal Garcia partitioning with 2 FDLs.*

3. PREEMPTIVE IMPLEMENTATION

A non-uniform partitioning consists of multiple FDLs which execute concurrently but perform their processing during different time periods. The shortest period, which should be equal to the audio callback interval, is associated with the primary FDL. Subsequent FDLs use larger block sizes and therefore have longer periods. In order to avoid having to process longer FDLs within a single callback interval, Gardner suggests that longer FDLs be allowed to run for a time interval equal to their period, rather than within a single callback period. This helps to preserve uniform processor loading. Figure 4 illustrates processing boundaries in time (arrivals and deadlines) for a partitioning with 3 FDLs.

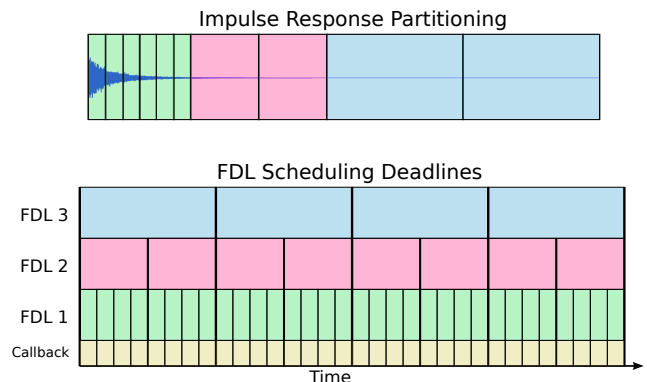


Figure 4: *Top - Example non-uniform partitioning with 3 FDLs. Bottom - Scheduling boundaries of FDL tasks. Arrivals/deadlines are denoted by vertical lines.*

3.1. Computation

In order to maintain a dropout free audio stream, we must ensure that all of an FDLs computations are complete by its processing deadline. In our case, the bulk of the processing time is spent computing FFTs and performing complex arithmetic. A popular choice for a portable FFT library is FFTW[8], which performs well on a wide variety of platforms by performing an “auto-tuning” step where it measures the performance of many FFT sub-routines and picks the fastest combination for the target system.

For longer FDLs (those containing more partitions), the complex multiply-add (Cmadd) routine becomes the computational bottleneck. We implemented an optimized version of the Cmadd routine, taking full advantage of Intel's SSE/SSE3 SIMD instructions and using regular memory access patterns. Our optimized Cmadd

code performed between 2.5x and 8.5x faster on our test platforms than a naive implementation that used GCC's built in complex types.

To put the absolute execution times of these routines in perspective, for an audio I/O buffer size of 32 samples at 44.1kHz, the callback interval is 725 μ s. A first level FDL will do one FFT and one inverse FFT each of size 64, and it will run the Cmadd routine once for every partition in the FDL. On a MacBook Pro with a 2.66GHz Intel Core 2 (Penryn) processor, FFTW's out-of-box forward and reverse FFT routines take 1.1 μ s and 0.32 μ s respectively, while the routines chosen using "FFTW_PATIENT" auto-tuning take 0.27 μ s and 0.25 μ s. The naive Cmadd routine takes 0.21 μ s per partition, while the optimized routine takes 0.087 μ s. At this block size, the optimized Cmadd routine allows us to handle almost three times as many partitions with the same processor load.

Finally, buffering operations in our Linux implementation greatly benefited from the use of *asmlib*[9], which includes optimized memory movement routines (*memset*, *memcpy*, etc) which can perform up to 10x faster than the default versions used by *glibc* for properly aligned memory regions.

3.2. Scheduling

As shown in Figure 4, the FDLs run concurrent tasks with different execution periods and deadlines. Because of this, we run each FDL in its own thread, which allows an FDL with an earlier deadline to preempt one with a later deadline. However, as we will discuss later, the inclusion of preemptive multi-threading within this implementation can cause problems when sharing computing resources with other audio processing tasks.

In our implementation, we use the POSIX threads (*pthread*s) API[10], to create and manage the execution and scheduling of worker threads. We create a worker thread for each FDL that is responsible for executing the FFTs and complex arithmetic required by the FDL. Synchronization of the worker threads and buffering/mixing operations are performed in the audio callback thread, which has the highest priority in the system and should preempt any other running threads. Since the primary FDL has the same period as the callback, we have the option of running it in the callback thread to avoid unnecessary context switches.

In order to get the worker threads to respect each others real-time deadlines, we use the "SCHED_FIFO" real-time scheduling policy with higher priorities assigned to FDLs with shorter periods. This policy allows lower priority threads to be preempted by higher priority threads. Effectively, this implements an *Earliest Deadline First* scheduler using the built-in scheduling mechanisms of common operating systems. It is important to make sure the callback thread is assigned the highest priority.

3.3. Thread Synchronization

Thread synchronization mechanisms typically rely on operating system calls, which can adversely affect real-time performance due to the variability in their execution times. Because of this, optimizing the synchronization between threads yielded the most significant performance improvements to our preemptive implementation.

We use two basic synchronization tasks in this implementation. In the first sync task, the main thread sends signals to worker threads telling them when to start. In the second, the worker threads signal the main thread when they are done. To implement these

operations, we use condition variables (*condvars*) and mutex locks from the *pthread*s library. Condition variables provide a mechanism for a thread to sleep until it receives a signal from another thread, and mutexes enable a thread to lock a shared memory region to prevent other threads from simultaneously accessing it. Signaling and waiting on a *condvar* require system calls, as does locking and unlocking a mutex, so we would like to minimize our use of these routines.

In a naive approach to these sync tasks, each of the worker threads would have their own *condvar* to wait on, and the main thread would have its own *condvars* to wait on (one for each worker thread). The main thread would signal each of the worker threads that need to be started during the current callback via their *condvars*. Then the main thread would wait on the *condvars* that correspond to the worker threads that have deadlines during the current callback. Worker threads communicate their completion by signalling the corresponding *condvar* belonging to the main thread. The problem with this approach is that if we have T worker threads, the main thread may need to send up to T *condvar* signals or wait on up to T *condvars* during a single callback. Each of these operations take between 3 μ s and 40 μ s (not including actual waiting time), so a few *condvar* ops could fill a significant portion of the callback period, leaving no time for actual computation or audio I/O.

In order to reduce the number of *condvars* required, we can reorganize the synchronization so that all worker threads that need to be started during a single callback are waiting on a single, specific *condvar*. Then only a single broadcast *condvar* signal is needed from the main thread to start the workers. Likewise, the main thread can wait on a single *condvar* that is signalled by the required worker thread that is last to finish. The problem with this approach is that in order for the worker threads to keep track of which is last to finish, they must all access a shared counter. If we protect this counter with a mutex, lock contention between the threads is introduced which can significantly stall their completion.

To remedy this lock contention problem, we can use atomic operations (we use the *gcc* builtin routines [11]), which remove the need for locks, do not require system calls, and are many orders of magnitude faster than mutex operations. If the worker threads atomically increment the shared counter when they complete, changing the value of the counter and getting its new value appear to occur instantaneously, negating the need for locks. This guarantees that the last thread to increment the counter will see the target counter value and know that it should send the completion signal to the *condvar* of the main thread. In addition, using atomic ops here allows the main thread to simply check if the counter has reached its target without acquiring a mutex before waiting on a *condvar*.

This method allows us to get close to 100% CPU utilization on a single core without dropouts. Figure 5 shows how this scheduling and synchronization approach works on a machine with 3 processor cores for a partitioning that uses 3 FDLs with the first FDL executing in the main callback thread. The period of the second FDL is twice that of the callback, and the third FDL has a period four times that of the callback.

3.4. Processing Multiple Channels

If we simply duplicated the scheduling and synchronization techniques outlined above for every channel when processing multiple channels, we would end up with a lot of redundant synchroniza-

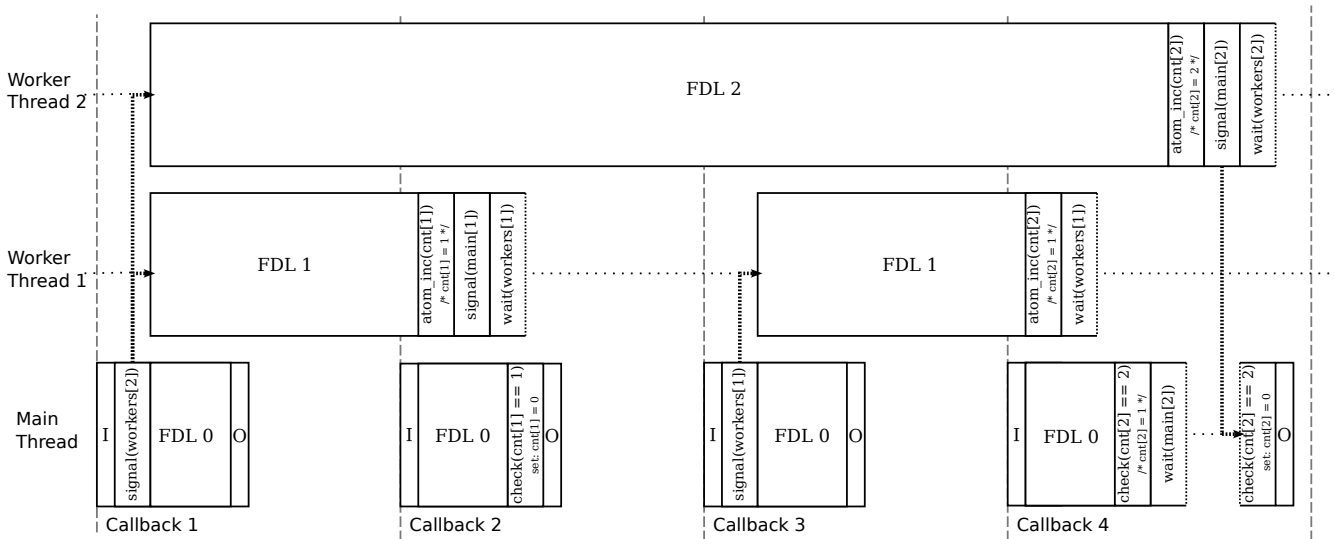


Figure 5: Synchronization walkthrough for a 3 FDL partitioning running on 3 cores. **I** = Input buffering, **O** = Output mixing

tion and probably a lot more threads than processor cores. To avoid this, FDLs belonging to different channels but with the same block size can be run in the same thread, since these FDLs all have the same arrivals and deadlines. Then the synchronization operations are shared amongst the channels, there is no extra synchronization overhead, and we keep the thread count low.

3.5. Targeting Multi-Core Architectures

The methods described above work fine when running on a single processor, because the thread priority assignments discussed in Section 3.2 will grant the processor to the thread with the most imminent deadline. When we have more than one processor core to work with, we can decide which core each thread should run on. Normally, the operating system will decide this for us, but this can yield suboptimal performance.

In Linux, we have the option of pinning threads to specific cores using non-portable (NP) extensions to the POSIX threads API. If we have at least as many cores as FDL levels, we could pin each FDL thread to its own core. This should minimize the number of preemptions and context switches. If we don't have enough cores to do this, we could still achieve a significant reduction in context switching by distributing the FDL threads evenly across the cores.

When processing multiple channels, another approach would be to create multiple worker threads per FDL level. This allows us to put the work belonging to a subset of the channels on each core, which would yield better load balancing across cores and possibly better memory locality. Because we would still be running all FDL levels on each core, there would still be lots of context switching, as in the single core case. We report on the performance of these thread pinning approaches in Section 5.

3.6. Choosing the Partitioning

After making all the low-level computational and scheduling decisions, we still have to decide how we will partition our impulse response(s). The approach in the Gardner paper is to double the

block size every two partitions[1], but this approach fails to take advantage of FFT/IFFT reuse within FDLs. Garcia has proposed a dynamic programming algorithm that determines an optimal partitioning in terms of number of mathematical operations[?]; however, this method fails to take into account actual execution time on the target system. The actual execution times of FFTs and Cmad routines operating on variably sized arrays can vary widely across hardware architectures and software implementations. This is why we feel it is important to measure the actual performance of the FDL work we are doing when choosing a partitioning, not unlike FFTWs "auto-tuning" stage.

Since we have real-time constraints, we must consider not the best-case or average performance of our FDLs but the worst-case performance. We estimate the worst-case performance of an FDL of a certain block size and number of partitions by polluting the L2 cache of our target machine before measuring the execution time of the FDL.

To determine a best partitioning from these performance numbers, we search for the valid combination of FDLs that has the lowest overall worst-case processor load. The processor load of each FDL is calculated by dividing its worst-case execution time by its period. We found that it was unnecessary to search the entire FDL space since — for a specific block size — the search would always choose the minimum number of partitions required by the block size of the subsequent larger FDL; therefore, the number of partitions in each FDL was restricted to powers of two.

4. TIME-DISTRIBUTED IMPLEMENTATION

An alternative implementation strategy for non-uniform partitioned convolution is to perform all the necessary computation within a single thread, manually partitioning the work such that the processing load is evenly distributed across processing frames. This means that during each frame, in addition to doing the processing for the smallest FDL, we also perform a fraction of the processing for each larger FDL. This approach requires significantly more programmer effort and a deeper understanding of the underlying mathematics than the previously described preemptive ap-

proach, since in this case we can't rely upon external libraries (i.e. FFTW) to do all the computational "heavy lifting." Our time distributed implementation is more constrained than our preemptive implementation and only supports two FDLs, with the secondary FDL being 32x the size of the primary FDL. However, the time distributed implementation has the benefit of fitting the existing model of plugins executing within an audio host application, where a plugin is expected to do its real-time processing within the context of a single high priority thread. Currently none of the audio host applications we are aware of provide mechanisms for plugins to create and schedule the execution of additional high priority threads. Our time distributed implementation utilizes a technique described by Hurchalla to uniformly divide the work associated with a secondary FDL across multiple frames. We summarize the technique here, for further details see [3].

FFT based block convolution involves three steps: a forward FFT of an input sequence, a complex multiplication of the resulting FFT coefficients with those of a stored impulse response, and an inverse FFT to generate an output sequence. One way to partition the work of performing block convolution with a given block size into smaller chunks of work is to use decimation in frequency (DIF) to divide the input and impulse response sequences into multiple subsequences. The FFT coefficients of the resulting subsequences are a subset of the FFTs of the original sequences. Our implementation uses two stages of radix-4 DIF to decompose the input sequence to the secondary FDL into 16 subsequences. During each frame of processing, we take the FFT of one of the input subsequences and perform half of the complex multiplications with the impulse response FFT coefficients. During a subsequent frame, we perform the second half of the complex multiplications and take the inverse FFT of the resulting values. This enables us to evenly distribute the FFT, complex multiplication, and inverse FFT steps across 32 processing frames. We use FFTW to perform the "leaf-level" FFT calculations. During each frame we also do a portion of the forward and inverse DIF decomposition for the previous block of input and the current block of output. This processing is not perfectly distributed across frames, resulting in a slight imbalance of the work done from frame to frame. The measured variation in execution time across frames for our implementation is less than 5 percent.

Hurchalla describes how to apply 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-4 decomposition steps, were hand optimized in assembly using the SSE extensions to the x86 ISA.

5. RESULTS

In this section we evaluate and compare the performance of our preemptive and time distributed implementations of partitioned convolution. The machine used to perform the benchmarking was a Mac Pro with two 2.66 GHz 6-core Intel Xeon "Westmere" processors and 12GB of memory running Linux 2.6.35 with low-latency realtime patches applied. A 10-channel ethernet audio

interface[12] was used for I/O. All experiments were performed at a sample rate of 44.1 kHz using a 64 sample frame size. The impulse response lengths we considered range from 16,384 – 524,288 samples (0.4 – 11.9 seconds). 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.

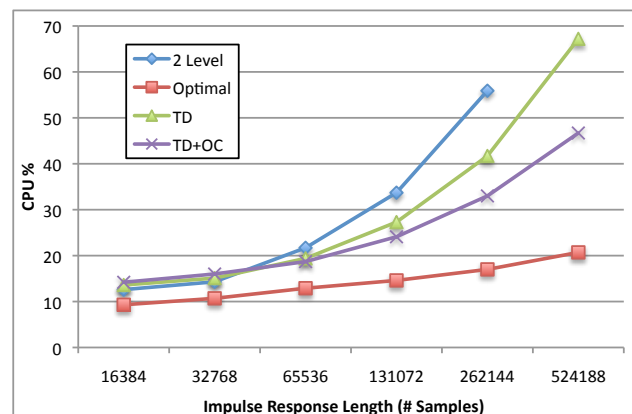


Figure 6: CPU usage on a single core.

For our first experiment, we disabled all but a single core in the system and recorded the reported CPU utilization for four different configurations executing the same workload. The workload was 16 independent channels of partitioned convolution, and the four configurations were: time distributed, time distributed with the 3-partition acyclic convolution optimization applied, preemptive using two FDLs, and preemptive using optimal partitioning (ranging from 3–5 FDLs). The results are presented in Figure 6. The time distributed and preemptive two-level implementations are using the same partitioning scheme (two FDLs of sizes 64 and 2048) so we might expect them to be similar in terms of computational load. This is true for the smaller partition sizes, but at the larger partition sizes (and more computationally intensive workloads) the time distributed implementation appears to have a clear advantage. This is at least partially attributable to the overhead of preemption and context switching: each context switch requires a trap into supervisor mode and the operating system kernel which incurs significant overhead. The time distributed implementation with the 3-partition convolution optimization outperforms the standard time distributed implementation, but the preemptive version using the optimal partitioning scheme outperforms all others by a wide margin.

Our second experiment was to measure how many instances (independent channels of convolution) each implementation was capable of running without experiencing any missed deadlines or dropouts. This experiment was also performed using only a single core. We increased the number of instances until we reached the highest point that ran dropout-free for 60 seconds. Figure 7 illustrates the results, which are quite different from what the previous experiment suggested. In this case, the two-level preemptive implementation is able to achieve more concurrent instances without dropouts than the time distributed version. We believe this is due to the greater regularity and predictability of the memory access patterns in the preemptive code — instead of performing only a portion of the computation related to the secondary FDL during

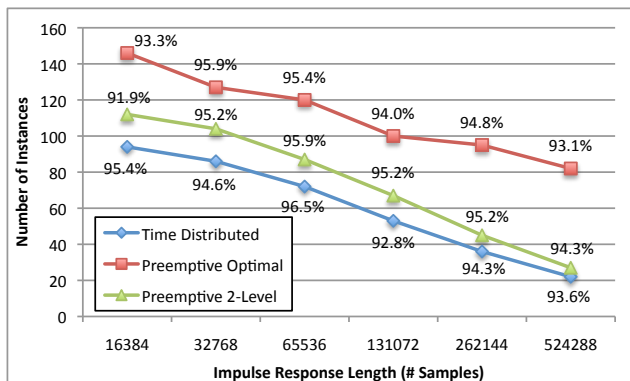


Figure 7: Max instances on a single core.

each sample frame, the preemptive version processes the FDLs for each instance to completion serially. This results in long streams of memory accesses with a constant stride, and the code is therefore able to benefit from the hardware prefetching mechanisms in the memory hierarchy to reduce the latencies cause by cache misses. Once again, the preemptive implementation using the optimal partitioning scheme is the clear winner, outperforming the others by a factor of 4x for the longest impulse response.

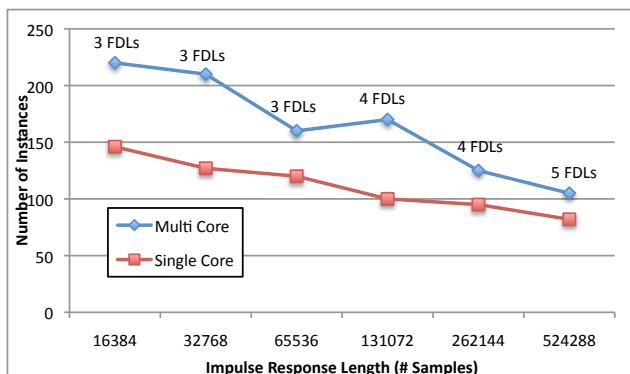


Figure 8: Max instances single core vs multi-core.

Our final experiment was to benchmark the preemptive implementation running on multiple cores. We assigned a single thread to process each FDL and pinned each thread to its own core to minimize disturbances from the OS scheduler. Since the optimal number of FDLs varies with impulse response length, so do the numbers of cores we used in these experiments. As mentioned in Section 3.5, we also considered an alternative scheme where channels (instead of FDLs) were distributed amongst the cores. In this case, 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 measurements, so we only present the results from the former here. A plot comparing the performance of the code running on single and multiple core configurations is presented in Figure 8. By using additional cores, we were able to run between 1.3x and 1.7x more instances without experiencing dropouts. While our work partitioning scheme is most likely not optimal (there is a significant difference in the

computational load across FDLs), we believe that ultimately the factor that limits the maximum achievable number of independent instances is memory bandwidth, not computational crunch. The processor used for these experiments has 12MB of last level cache and 256KB of private level 2 cache per core. A 524,288 sample impulse response represented as single precision floating point values occupies 2MB of memory. Clearly for the large number of concurrent instances we are able to run, the working set doesn't fit into the on-chip cache and the latency of DRAM accesses becomes the bottleneck for achievable performance.

6. DISCUSSION

In all the scenarios we investigated, the preemptive implementation of partitioned convolution using an empirically determined optimal partitioning outperformed all the others by a wide margin. Our motivation for implementing the time distributed version was to be able to use it in the context of an audio processing environment such as Max/MSP or Pd. However, this is just a stop-gap solution and in the future we hope that audio host applications will provide mechanisms for plugins or objects to schedule their execution across multiple concurrent threads. Another advantage of the preemptive approach lies in the programmer effort required to implement it. While efficiently managing the scheduling of multiple threads is not trivial, it affords us the opportunity to use existing highly optimized libraries to perform necessary computations without needing to worry about manually partitioning the work. Optimizing the time distributed FFT to the point that it was competitive with FFTW's FFT routines required hand tuning assembly code and carefully managing data layout which was an arduous task. Also, the techniques used to implement the time distributed FFT don't scale well to larger (greater than 32x) FDL partition sizes, which limits the performance of the time-distributed partitioned convolution algorithm for very long impulse response lengths.

6.1. Further optimizations

6.2. The need for Preemption and Multi-threading

Partitioned convolution is but 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 the programmer to time-distribute long-running tasks evenly across multiple short time periods, particularly when those tasks call external libraries. For this particular case (FFTs) there are clever tricks that allowed us to accomplish this in a limited manner but other computations (for example, those related to machine learning algorithms) may not be as amenable to such treatment.

While it is possible for us to spawn worker threads and attempt to manage them, even while running in the context of existing audio host applications, there is no guarantee that other plug-ins running on the host won't do the same thing. This would result in pollution of our "thread ecosystem" and force our threads to compete with others for processor time and cache space. Ultimately when there are more threads than cores in the system, the responsibility for scheduling the threads falls onto the operating system, which can only do so well given that it has very limited knowledge about the relationships and dependencies between threads. If the audio host itself were able to manage its hardware resources

(processor cores and caches) by arranging the execution of plug-in tasks on multiple cores then plug-in programmers and end users could benefit from improved parallel performance, programmability, and quality of service. On-going research into the development of a research operating system that enables applications with real-time constraints running on parallel machines to more explicitly schedule the execution of their constituent threads is described in [13].

6.3. Final Remarks

always end with a quote

7. REFERENCES

- [1] WG Gardner, “Efficient convolution without input-output delay,” *Journal of the Audio Engineering Society*, vol. 43, no. 3, pp. 127–136, 1995.
- [2] Dave Cohen et al., “Efficient simultaneous multi-scale computation of ffts,” Tech. Rep. FODAVA-09-01, Georgia Tech, Aug 2009.
- [3] J Hurchalla, “A time distributed FFT for efficient low latency convolution,” *Audio Engineering Society Convention 129*, 2010.
- [4] J Hurchalla, “Low latency convolution in one dimension via two dimensional convolutions: An intuitive approach,” *Audio Engineering Society Convention 125*, 2008.
- [5] A. Oppenheim and R. Schaffer, *Discrete-Time Signal Processing*, Prentice Hall, Englewood Cliffs, 1999.
- [6] Dave Cohen, “Efficient simultaneous multi-scale computation of ffts,” *FODAVA-09-01*, pp. 1–3, Aug 2009.
- [7] D Orozco, L Xue, M Bolat, X Li, and G.R Gao, “Experience of optimizing FFT on Intel architectures,” *2007 IEEE International Parallel and Distributed Processing Symposium*, p. 448, 2007.
- [8] Matteo Frigo and Steven G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, Special issue on “Program Generation, Optimization, and Platform Adaptation”.
- [9] Agner Fog, “Instructions for asmlib: a multi-platform library of highly optimized functions for C and C++,” <http://www.agner.org/optimize/asmlib-instructions.pdf>, 2010.
- [10] Bradford Nichols, *Pthreads Programming: a Posix Standard for Better Multiprocessing*, O’Reilly, Sebastopol, 1996.
- [11] Using the GNU Compiler Collection (GCC), “Built-in functions for atomic memory access,” <http://gcc.gnu.org/onlinedocs/gcc/Atomic-Builtins.html>, 2011.
- [12] Rimas Avizienis, Adrian Freed, Takahiko Suzuki, and David Wessel, “Scalable connectivity processor for computer music performance systems,” in *Proc. of the International Computer Music Conference*, Berlin, Germany, 2000, pp. 523–526, International Computer Music Association.
- [13] Colmenares et al., “Real-time musical applications on an experimental operating system for multi-core processors,” in *Proc. of the International Computer Music Conference (awaiting acceptance)*, Huddersfield, England, 2011.