Efficient Convolution Without Latency

William G. Gardner
Perceptual Computing Group
MIT Media Lab E15-401B
20 Ames St
Cambridge MA 02139-4307
Internet: billg@media.mit.edu

November 11, 1993

Abstract

It is well known that a block FFT implementation of convolution is vastly more efficient than the direct form FIR filter. Unfortunately, block processing incurs significant input/output latency which is undesirable for real-time applications. A hybrid method is proposed for doing convolution by combining direct form and block FFT processing. The result is a zero latency convolver that performs significantly better than direct form methods.

Introduction

In various audio applications, there is a need to perform large finite impulse response (FIR) filters. One such application is room reverberation simulation, in which a synthetic or sampled room response is convolved with a source sound to simulate the room. Typical room responses can span several seconds, and thus a two second room impulse response will require a 88,200 point FIR filter at a 44.1 kHz sampling rate. Clearly, implementing such a filter using the direct form (multiply accumulate) method is impractical. It is vastly more efficient to use a block transform method based on the fast Fourier transform (FFT), such as the overlap-add or overlap-save methods [1,2]. These methods collect a block of input samples, transform them into the frequency domain, multiply by the spectrum of the filter response, and inverse transform to obtain output samples. Unfortunately, a block transform implementation will cause significant input/output latency, because you must wait for an entire block of input samples to accumulate, and then perform significant calculations before an output sample is generated. Let's say we have a processor which convolves using 128K point blocks (thus it can render a 3 second impulse response at a 44.1 kHz sampling rate). Every 3 seconds, the processor generates 3 seconds of output by transforming the input, multiplying by the response, and inverse transforming. If it stays constantly loaded repeating this operation, we would expect the input/output latency to be 6 seconds: 3 seconds to wait for a block of input samples, and 3 seconds to perform the calculation. This is an eternity for real-time applications, such as live performance, or a post-production mixing application.

There is a simple way to harness the efficiency of block transform methods without sacrificing latency. The linearity of convolution allows us to break up the filter response into blocks of different size, compute the convolution of each block with the input signal, and sum the results to obtain the output signal. We can render the initial portion of the filter response using a direct form filter, which buys us enough time to accumulate a small block of samples and render the next portion of the filter response using a block transform method. This in turn buys us additional time to perform longer block transforms to render further delayed portions of the impulse response. Essentially, early portions of the impulse response are rendered using small block transforms, and late portions of the impulse response are rendered using large block transforms. To achieve zero latency, the earliest portion of the filter response must be rendered using a direct form filter. Of course, when we say zero latency, we do not include the one or two sample latency usually inherent in DSPs due to input/output buffering.

This paper will propose such a method for doing efficient, zero latency convolution. The method requires doing both a direct form filter and sets of various sized block FFT convolutions. Results of the small block transforms can be used to optimize calculating the larger block transforms. The design problem is how to split up the long filter response into smaller portions, and how to manage the tasks so that everything is done on time and the processor is fully utilized. We propose using a primitive process scheduler to manage DSP tasks. We will compare our method to both direct form and large block transform methods. As expected, our method is far more efficient than direct form implementation, but is much less efficient than large block, large latency methods. With our method, a small amount of fixed latency can be traded for additional efficiency.

Some assumptions have been made. In order to assess the cost of an implementation we will count the number of real multiplications and additions required. This obviously misses lots of processing which must occur, but we feel that any more detailed analysis of the algorithms is useless since execution speed will be completely dependent on details of the target DSP. We will also restrict ourselves to using radix-2 FFTs (i.e. block sizes which are a power of two) to implement block transform convolution.

Notation

We will introduce some notation to make the discussion clearer. First, we need a way to specify subsequences of other sequences. Let's define x(n;N) to be the N point subsequence of the sequence x starting at index n, thus x(n;N)[k] = x[n+k], for 0 <= k < N. Let's also define a concatenation operator | which concatenates two sequences, thus x(0;N) | x(N;M) = x(0;N+M). Finally, we need a block convolution operator that calculates one block of a convolution result according to the overlap-save method of block convolution. Let y = x * h, where * denotes convolution and h is an N-point sequence, and define the operator • such that x(n;N) • h = y(n;N).

The block convolution operator • takes two N point arguments and yields an N-point result, but to do the calculation requires knowing 2N points of the first argument. In the calculation x(n;N) • h, we need to know the N points of x prior to index n, because these affect y(n;N). To do the calculation, we form the 2N-point sequence of x(n-N;N) concatenated with x(n;N), which is simply x(n-N;2N), and we calculate the discrete Fourier transform (DFT) of this sequence. Then we form the 2N-point sequence of h concatenated with N zeroes, and calculate the DFT of this sequence. Then we multiply the two spectra, and compute the inverse DFT of the product. This yields a 2N-point result and we keep the last N points of this result. The first N points will in general be time-aliased and we throw these out. We can write this calculation as follows:

$$x(n{:}N) \bullet h \;=\; IDFT\{DFT\{x(n-N;2N)\} \cdot DFT\{h \mid z(0;N)\}\}(N;N) \qquad (1)$$

where h is an N-point sequence and z is a sequence of zeroes. It follows directly from the definition of the block convolution operator that the convolution y = h * x can be expressed as the concatenation of results of the block convolution operator:

$$y \;=\; x{*}h \;=\; x(0;N){\bullet}\, h \mid x(N;N) \bullet h \mid x(2N;N) \bullet h \mid \ldots \mid x(kN;N) \bullet h \qquad (2)$$

This is simply a formalization of the overlap-save method.

Minimum computation cost solution

It is easy to divide the filter response into blocks such that we minimize the total computation cost per output point without violating the zero latency constraint. This is done by using the largest possible blocks to calculate the convolution. Figure 1a shows a minimum cost solution to the zero latency problem.

<insert figure 1>

The filter response h is broken into blocks as shown in the figure, such that $h_0 = h(0;N)$, $h_1 = h(N;N)$, $h_2 = h(2N;2N)$, and $h_3 = h(4N;4N)$. The first block, h_0, is N points long and its result, y_0, is rendered via direct form filtering without latency. For small filter lengths, direct form filtering is more efficient than the block transform method. Here, N is the size at which block convolution becomes more efficient than direct form filtering; for a typical DSP this might be 32 or 64 samples. We will refer to N as the starting block size, which is defined as the block size at which block convolution becomes more efficient than direct form filtering. The computation of y_0 via direct form filtering allows us just enough time to accumulate a block of N input samples to calculate $y_1 = x * h_1$ using an N-pt block transform. As should be evident, this calculation must be completed within one sampling period, or we will not produce the result y_1 in time. Continuing with this strategy, the calculations of y_0 and y_1 allow us just enough time to accumulate a 2N-pt block of input samples to calculate $y_2 = x * h_2$, which is done using 2N point blocks. The last calculation shown is $y_3 = x * h_3$, which is done using 4N-point blocks. We would increase the filter response size be adding blocks of size 8N, 16N, 32N, etc. With this scheme, each block of size M starts M samples into the filter response h (except for the direct form block h_0). We can see that the result is a minimum cost solution because there is no way to increase any block size without violating the latency constraint, and the larger block transforms require fewer calculations per output sample.

Although this is a minimum cost solution in terms of computation per output sample, it is completely impractical. This is because all the computation must be done during short periods of time. Figure 1b depicts processor demand as a function of time. The computation of y0 by direct form filtering requires constant processor time per output sample. The computation of y_1, on the other hand, requires a burst of activity every N samples. All the computation must be done within one sampling period. Similar constraints are placed on the calculation of y_2 and y_3. For instance, at time 4N, we have to do an N-pt, 2N-pt, and 4N-pt calculation, all within one sampling period. Even if we had a processor fast enough to do this, we would be wasting its power, since most of its time would be spent waiting for a block of samples. Thus, there is an implicit constraint that the processor should be fully utilized, and this can be met by distributing processor demand evenly over time.

A constant demand solution

We want to avoid the situation where a repetitive calculation must complete in less time than the period of repetition. This causes the demand on the processor to be non-uniform. Let's say we have broken the filter reponse into an M+P point direct form filter and an M point block convolution, where $P < M$, as shown in figure 2.

<insert figure 2>

The M point block convolution is a repetitive calculation which must be done every M samples, and which must complete in P samples. If we allocate just enough processor bandwidth to complete this calculation in P samples, then there are M-P samples during which this processor bandwidth is not being used. We can try to fill this free time with another task such that the overall processor usage is constant. Since this free time slot occurs periodically every M samples, we can fill the free time perfectly only by another task which can itself be uniformly distributed over time. Thus, we still need a way to uniformly distribute processor load over time <weak argument!>.

One way to ensure constant processor loading is to have blocks of size M start at least 2M points into the filter response. This way, there are at least M sample periods to let the block accumulate and M sample periods to do the calculation. Since the calculations need to be done every M sample periods, this keeps the processor evenly loaded over time. Figure 3 shows how to break up the filter response so the processor demand stays constant.

<insert fig 3>

In the diagram, the first 2N points of h are rendered with a direct form filter, and the remaining blocks of h are of sizes N, N, 2N, 2N, 4N, 4N, etc. We have labelled the filter response blocks as hn, where h_0 = h(0;2N), h_1 = h(2N;N), h_2 = h(3N;N), etc. Note that each block of size M starts at least 2M points into the filter response (except for h_0). The processor demand diagram shows when each block operation can begin and its completion deadline. For instance, the block operation x(0;N) • h_1 can begin at time N, and must complete by time 2N. Similarly, the block operation x(0;N) • h_2 can begin at time N, but does not need to complete until time 3N. The diagram clearly shows uniform distribution of processor demand.

Process scheduling

Without violating any of the deadline constaints, we can rearrange figure 3 as shown in figure 4.

<insert fig 4>

We can think of the periodic block convolutions with different parts of the filter response as separate processes, or tasks, which must be completed on time. We have separate tasks associated with h_0 through h_6, seven periodic tasks in all. Convolution with h_0 is special, in that a result must be obtained every sampling period via direct form filtering. Hence, convolution with h0 will be calculated during the sample interrupt procedure. When the processor is not servicing a sample interrupt, it continues processing one of the block convolution tasks. Every N sample interrupts we hit a block boundary, and we must schedule (start) the appropriate block tasks. The N-point blocks have the most immediate deadline, and thus should have the highest priority. The 2N-point tasks should have medium priority, and the 4N-point tasks should have the lowest priority. So for this example we have three priority levels of tasks that must be completed. The figure below shows what tasks are scheduled at times t = 4N, 5N, 6N, and 7N. This scheduling follows directly from figure 4.

time t = 4N:
- high priority: x(3N;N)•h_1, x(3N;N)•h_2
- medium priority: x(2N;2N)•h_3, x(2N;2N)•h_4
- low priority: x(0;4N)•h_5, x(0;4N)•h_6

time t = 5N:
- high priority: x(4N;N)•h_1, x(4N;N)•h_2

time t = 6N:
- high priority: x(5N;N)•h_1, x(5N;N)•h_2
- medium priority: x(4N;2N)•h_3, x(4N;2N)•h_4

time t = 7N:
- high priority: x(6N;N)•h_1, x(6N;N)•h_2

Figure 5. Task scheduling from figure 4.

At time t = 4N, we will schedule tasks at all three priority levels. The highest priority task is to calculate x(3N;N) • h_1 and then calculate x(3N;N) • h_2. We want to do these sequentially, because we can use the intermediate results of the x(3N;N) • h_1 calculation to make the x(3N;N) • h_2 calculation faster. In particular, the 2N-point DFT of x(2N;2N) is used in both calculations. At time t = 4N we also schedule two, 2N-point block calculations at medium priority, and two, 4N-point block calculations at low priority.

Process scheduler

The process scheduler can be very simple. Basically the scheduler maintains a list of tasks to be completed, in order of priority. The head of the list should be the highest priority task. When the processor is not servicing a sample interrupt, it returns to executing the highest priority task on the list. When this task completes, the next item on the list is executed. Every N samples, the interrupt task will schedule new block tasks to be completed. These must be added to the task list in sorted order of priority. If we see that

we are adding tasks to the list at the same priority as existing tasks, then we know we are not meeting the deadlines. This is because all the M-point block tasks should be complete before any new M-point tasks are scheduled.

We are always going to schedule high priority tasks every N samples. Presumably the processor was previously executing a lower priority task, so this is the time when the processor switches context from a low priority task to a high priority task. We need to remember the state of low priority task and save this information in its task control block in the task list. For a typical DSP (e.g. Motorola 56000), the state of a process consists of the machine stack, the status register and program counter, and all address and data registers. Since we are currently in an interrupt routine, the status register and PC are already on the stack, so we simply need to copy the stack and registers to the task control block. Then we can push the address of the process scheduler onto the stack, and execute a return from exception instruction which will return us to the process scheduler. The scheduler will in turn execute the high priority task that is first on the task list. High priority tasks are never interrupted by another task (except the sample interrupt task), so we never have to restore their context. However, all lower priority tasks will be interrupted, so when the process scheduler executes a lower priority task, it must first check to see if there is any context to restore. If so, then this is done by restoring the stack and registers, and executing a return from exception instruction which will restart the task as if we had returned from the sample interrupt that switched context in the first place. Obviously, there are many ways to do this sort of process scheduling, and implementation details will depend on the target processor.

Because the algorithm as a whole is data independent and time-invariant, we can expect the behavior of the process scheduler to be completely periodic. Thus, we could replace the process scheduler with something much simpler, but this would require splitting up the large block tasks into smaller tasks with known computational requirements. It seems easier and more flexible to have the process scheduler do all the work for us. With the scheduler approach, we can increase the size of the filter response in blocks until the algorithm overruns, then decrease the size of the response by one block so that it runs on time. There will probably be some free processor time left over, and we can increase the size of the response by a small block to fill this remaining processor time.

The zero latency convolution algorithm

We want to compare the computational cost of the zero-latency algorithm with the direct FIR approach and the large block, large latency approach. To do this, we need to specify an exact convolution algorithm and count operations. As a measure of computational complexity, we will count real multiplications and additions. A great deal has been written on methods to optimize calculation of the DFT, and related methods to optimize convolution [3,4,5,6]. Here, we will avoid exotic implementations, opting instead for convolution based on the commonly available radix-2 Cooley-Tukey FFT algorithm [7]. We have already introduced the block convolution operator • based on the overlap-save method. Of the two basic block convolution methods, overlap-add and overlap-save, overlap-save is more efficient because we do not have to do the additions required in the overlap-add method, and otherwise they require the same amount of computation. We will optimize the overlap-save method by using real input FFTs instead of complex FFTs, and observing symmetry when computing the spectral product. We will not consider any additional optimizations to the basic block convolution method. However, there are a number of optimizations that can be done in the context of the zero-latency scheme. In particular, the computation of the input spectra for a large block convolution can be based on the results from smaller blocks, or it can be reused directly.

Real DFT

It is well known that a N-point real input DFT can be done using an N/2-point complex DFT [6]. We will quickly go through the derivation so we can refer to it later. The basic idea is that using an N/2-point complex DFT we can do two, N/2-point real DFTs in parallel by assigning our two real sequences to the real and imaginary parts, respectively, of the complex input sequence. Because the real part of the input yields a conjugate-symmetric result, and the imaginary input yields a conjugate-antisymmetric result, we can separate the two results from the result of the complex DFT. If the two real sequences were the even and odd samples of an N-point real sequence, then the two real DFT results can be combined with a set of

butterflies to give the N-point real DFT result according to the decimation-in-time decomposition of the DFT.

The derivation is as follows (see [2], problem 9.31). Consider two, N-point, real sequences $x_1[n]$ and $x_2[n]$ with corresponding DFTs $X_1[k]$ and $X_2[k]$, respectively. Let g[n] be the complex sequence $x_1[n] + jx_2[n]$ with corresponding DFT $G[k] = G_R[k] + jG_I[k]$, and let $G_{OR}[k]$, $G_{ER}[k]$, $G_{OI}[k]$, and $G_{EI}[k]$ denote the odd part of the real part, the even part of the real part, the odd part of the imaginary part, and the even part of the imaginary part, respectively, according to:

$$G_{OR}[k] = \tfrac{1}{2}\{G_R[k] - G_R[N-k]\} \quad (3)$$

$$G_{ER}[k] = \tfrac{1}{2}\{G_R[k] + G_R[N-k]\} \quad (4)$$

$$G_{OI}[k] = \tfrac{1}{2}\{G_I[k] - G_I[N-k]\} \quad (5)$$

$$G_{EI}[k] = \tfrac{1}{2}\{G_I[k] + G_I[N-k]\} \quad (6)$$

Then $X_1[k]$ and $X_2[k]$ can be expressed as:

$$X_1[k] = G_{ER}[k] + jG_{OI}[k] \quad (7)$$

$$X_2[k] = G_{EI}[k] - jG_{OR}[k] \quad (8)$$

This shows how we would perform two, N-point real DFTs using a single N-point complex DFT. We can use the same strategy to perform two, N/2-point real DFTs using a single N/2-point complex DFT.

Now let x[n] be an N-point real sequence with DFT X[k] as follows:

$$DFT\{x[n]\} = X[k] = \sum_{n=0}^{N-1} x[n]W_N^{nk} \quad (9)$$

$$\text{where } W_N = e^{-j2\pi/N} \quad (10)$$

If we let $x_1[n] = x[2n]$ (even numbered samples of x), and let $x_2[n] = x[2n+1]$ (odd numbered samples of x), then we can express X[k] as shown below.

$$X[k] = \sum_{n=0}^{(N/2)-1} x[2n]W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x[2n+1]W_N^{(2n+1)k} \quad (11)$$

$$= \sum_{n=0}^{(N/2)-1} x_1[n]W_{N/2}^{nk} + W_N^k \sum_{n=0}^{(N/2)-1} x_2[n]W_{N/2}^{nk} \quad (12)$$

This is the formulation of the decimation-in-time FFT decomposition. The two summations represent the N/2-point DFTs of x_1 and x_2. The expression shows how the two, N/2-point DFTs are recombined via a set of butterfly operations to form the N-point DFT.

Cost of real DFT

We want to determine the amount of computation needed to do an N-point, real DFT in terms of real multiplications and additions. Unless explicitly stated otherwise, references to multiplications and additions will be assumed to be real. An N-point radix-2 complex DFT requires (N/2)log(N/2) complex multiplications and NlogN complex additions, where the logarithm is base 2. This does not count any

complex multiplications for one stage of butterflies (because these multiplications are all unity gain), but otherwise counts all complex multiplications regardless of whether they can be implemented without multiplication (or whether they can be done with a reduced number of operations). If complex multiplications are implemented using 4 multiplications and 2 additions (an alternative approach requires 3 multiplications and 3 additions [4]), this yields 2NlogN - 2N multiplications and 3NlogN - N additions for an N-point complex DFT. To perform an N/2-point complex DFT therefore requires NlogN - 2N multiplications and (3/2)NlogN - 2N additions. To perform an N-point real DFT, we need to do an N/2-point complex DFT plus the spectral separation according to equations 4-8 plus the butterfly operations of equation 12. Naively, the spectral separations would require N multiplications to scale by 1/2, and N additions each to form $X_1[k]$ and $X_2[k]$. We can skip the scaling multiplications, which will give us twice the expected result, and compensate by scaling the filter response by 1/2. Also, the symmetry of the spectra allows us to calculate only half of $X_1[k]$ and $X_2[k]$, which require N/2 additions each. The final set of butterflies would normally require N/2 complex multiplications and N complex additions, but we only need to calculate N/2 complex additions because the result is symmetric. Therefore, an N-point real DFT requires a total of NlogN multiplications, and (3/2)NlogN + N additions. This result is summarized in the table 1.

	multiplications	additions
N/2-point complex DFT	NlogN - 2N	(3/2)NlogN - 2N
spectral separation	0	N
spectral recombination	2N	2N
total	NlogN	(3/2)NlogN + N

Table 1. Computational cost of N-point real DFT.

Using a similar method, but paying more attention to optimization details, Sorenson [6] reports NlogN - (7/2)N - 6 multiplications and (3/2)NlogN - N/2 - 2 additions. We won't worry about the differences between the operation counts; it is the constants of proportionality of the NlogN terms which are ultimately important, and these factors agree. Any optimization which affects the N terms simply changes the cost per output sample by a constant amount regardless of the blocksize N.

Cost of block convolution operator

Now let's determine the computational cost of the block convolution operator given we have both the input signal spectrum and the filter response spectrum already computed. We will always be able to precompute the filter response spectrum, so we are not interested in this cost. We do need to calculate the input spectrum, and we can do this by directly computing it using the real DFT described above, or through other means described later. In either case, we will add the cost of this spectral computation to the block convolution cost to obtain a total cost.

For an N-point block convolution operation (•), we need to multiply two, 2N-point spectra and compute the 2N-point inverse real DFT. Computation of the spectral product requires only N complex multiplications since the spectra are symmetric, and we will assume the inverse real DFT requires the same number of computations as the forward transform. Therefore, the N-point block convolution operator requires NlogN + 4N multiplications, and (3/2)NlogN + 3N additions.

Reusing input signal spectra

In the context of the zero latency algorithm, there are additional optimizations that can be done. The most obvious is to reuse the spectrums of the input signal whenever possible. For instance, when we calculate $x(n;N) \bullet h_2$, we can use the input signal spectrum we obtained when calculating $x(n;N) \bullet h_1$. So the cost of doing these two calculations is the cost of a 2N-point real DFT plus two, N-point block convolution operations. This optimization can be applied to all equal sized blocks of the filter response.

Another significant optimization can be applied to calculating the spectrum of large blocks of the input signal. As an example, consider the calculation of x(2N;2N) • h_3. This requires that we calculate the spectrum of x(0;4N). We will already have the spectra for both x(0;2N) and x(2N;2N), because the scheduler will have already executed the tasks x(N;N) • h_1 and x(3N;N) • h_1. By adding the spectra of x(0;2N) and x(2N;2N), we can trivially obtain the even points of the spectrum of x(0;4N). Thus, we just need to obtain the odd points of the spectrum of x(0;4N). It turns out that by using the half-sized spectra to determine the even spectral points and calculating the unknown odd spectral points requires slightly more than half the computation of calculating the spectrum from scratch.

The DFT-odd computation

We will examine the method of obtaining the spectrum X[k] of an N-point sequence x[n] given that we have the spectra of two, N/2-point sequences $x_1[n]$ and $x_2[n]$ which consist of the first N/2 points and the second N/2 points of sequence x, respectively. Thus, $x_1[n] = x[n]$, $x_2[n] = x[n + N/2]$, for $0 <= n < N/2$. All the needed relationships fall out of the derivation for the decimation-in-frequency FFT [8]. The N-point DFT of x[n] is:

$$DFT\{x[n]\} = X[k] = \sum_{n=0}^{N} x[n] W_N^{nk} \tag{13}$$

$$= \sum_{n=0}^{N/2-1} x_1[n] W_N^{nk} + \sum_{n=0}^{N/2-1} x_2[n] W_N^{(n+N/2)k} \tag{14}$$

$$= \sum_{n=0}^{N/2-1} \left[x_1[n] + e^{-j\pi k} x_2[n] \right] W_N^{nk} \tag{15}$$

We now consider the even and odd points of the spectrum X[k].

$$X[k = 2r] = \sum_{n=0}^{N/2-1} \left[x_1[n] + e^{-j\pi 2r} x_2[n] \right] W_N^{n\,2r} \tag{16}$$

$$= \sum_{n=0}^{N/2-1} \left[x_1[n] + x_2[n] \right] W_{N/2}^{nr} \tag{17}$$

$$= \sum_{n=0}^{N/2-1} x_1[n] W_{N/2}^{nr} + \sum_{n=0}^{N/2-1} x_2[n] W_{N/2}^{nr} \tag{18}$$

$$= DFT\{x_1[n]\} + DFT\{x_2[n]\} \tag{19}$$

$$X[k = 2r + 1] = \sum_{n=0}^{N/2-1} \left[x_1[n] + e^{-j\pi(2r+1)} x_2[n] \right] W_N^{n(2r+1)} \tag{20}$$

$$= \sum_{n=0}^{N/2-1} \left[x_1[n] - x_2[n] \right] W_N^n W_N^{n2r} \tag{21}$$

$$= \sum_{n=0}^{N/2-1} \left\{ \left[x_1[n] - x_2[n] \right] W_N^n \right\} W_{N/2}^{nr} \tag{22}$$

Equation 19 shows that the even points of X[k] can be determined by adding the spectra $X_1[k]$ and $X_2[k]$. Equation 22 shows that the odd points of X[k] can be determined by pre-aliasing and modulating x[n], and then performing an N/2-point complex DFT. Unfortunately, this doesn't yield any savings, because we already have a method of calculating X[k] using an N/2-point complex DFT since x[n] is real. We will show how the calculation of equation 22 can be done using an N/4-point complex DFT when x[n] is real. This is exactly analogous to the method used to compute an N-point real DFT using an N/2 point complex DFT.

First, let's redefine equation 22 and call it the "DFT-odd" operation:

$$DFT-odd\{x[n]\} = X[k=2r+1] = X'[r] = \sum_{n=0}^{N-1} x[n] W_N^{n(2r+1)} \tag{23}$$

for 0 <= r < N/2. If x[n] is real, then X'[r] will be symmetric as follows:

$$X'[r] = X'^*[(N/2)-r-1] \tag{24}$$

If x[n] is pure imaginary, then X'[r] will be antisymmetric as follows:

$$X'[r] = -X'^*[(N/2)-r-1] \tag{25}$$

Note that this is slightly different than the usual form of spectral symmetry because these are the odd points of a conjugate symmetric spectrum. The symmetry allows us to perform two N-point real DFT-odd operations using a single N-point complex DFT-odd operation.

The derivation exactly follows the real DFT derivation except for the form of the symmetry. Consider two N-point real sequences $x_1[n]$ and $x_2[n]$ with corresponding N/2-point DFT-odd transforms X1'[r] and X_2'[r], respectively. Let g[n] be the complex sequence $x_1[n] + jx_2[n]$ with corresponding DFT-odd G'[r] = G_R'[r] + jG_I'[r], and let G_{OR}'[r], G_{ER}'[r], G_{OI}'[r], and G_{EI}'[r] denote the odd part of the real part, the even part of the real part, the odd part of the imaginary part, and the even part of the imaginary part, respectively, according to:

$$G_{OR}'[r] = \tfrac{1}{2}\{G_R'[r] - G_R'[(N/2)-r-1]\} \tag{26}$$
$$G_{ER}'[r] = \tfrac{1}{2}\{G_R'[r] + G_R'[(N/2)-r-1]\} \tag{27}$$
$$G_{OI}'[r] = \tfrac{1}{2}\{G_I'[r] - G_I'[(N/2)-r-1]\} \tag{28}$$
$$G_{EI}'[r] = \tfrac{1}{2}\{G_I'[r] + G_I'[(N/2)-r-1]\} \tag{29}$$

Then X_1'[r] and X_2'[r] can be expressed as:

$$X_1'[r] = G_{ER}'[r] + jG_{OI}'[r] \tag{30}$$
$$X_2'[r] = G_{EI}'[r] - jG_{OR}'[r] \tag{31}$$

Let's return to the DFT-odd formulation of equation 23 and consider summing over the even and odd samples of x[n]. If we let $x_1[n] = x[2n]$ (even numbered samples of x), and let $x_2[n] = x[2n+1]$ (odd numbered samples of x), then we can express X'[r] as shown below.

$$X'[r] = \sum_{n=0}^{N-1} x[n] W_N^{n(2r+1)} \tag{32}$$

$$= \sum_{n=0}^{(N/2)-1} x[2n] W_N^{2n(2r+1)} + \sum_{n=0}^{(N/2)-1} x[2n+1] W_N^{(2n+1)(2r+1)} \tag{33}$$

$$= \sum_{n=0}^{(N/2)-1} x_1[n] W_{N/2}^{n(2r+1)} + W_N^{2r+1} \sum_{n=0}^{(N/2)-1} x_2[n] W_{N/2}^{n(2r+1)} \tag{34}$$

The summations in equation 34 are recognized as the N/2-point DFT-odd transforms of $x_1[n]$ and $x_2[n]$. The expression shows how to combine the two N/2-point transforms to create the N-point transform using a set of butterfly operations.

In order to compute the N-point DFT-odd transform of a real sequence x[n], we form the N/2-point sequences consisting of the even and odd samples of x[n] and assign these to the real and imaginary parts, respectively, of an N/2-point complex sequence. We then compute the DFT-odd of this sequence using equation 22, which only requires doing an N/4-point complex DFT. We then perform a spectral separation according to equations 26-31, and finally we perform the spectral recombination using a set of butterflies according to equation 34.

Cost of real DFT with prior results

Let's calculate the computational cost of determining the real DFT of an N-point sequence x assuming we have the N/2-point DFTs of the first N/2 points and the second N/2 points of x. To obtain the N/2 even spectral points requires only N/4 complex additions due to symmetry. To calculate the odd points requires doing an N/2-point complex DFT-odd transform. The pre-aliasing requires N/4 complex additions, and the modulation requires N/4 complex multiplications. Then, the N/4-point complex DFT requires (N/2)logN - (3/2)N multiplications and (3/4)NlogN - (7/4)N additions. The spectral separation requires N/2 scaling multiplications and N/2 additions. The final butterflies require N/4 complex multiplications and N/4 complex additions. The total cost is (N/2)logN + N multiplications and (3/4)NlogN + (5/4)N additions. This is slightly more than half the cost of calculating the real spectrum from scratch, so we are really saving significant computation by using this approach, despite its apparent complexity. These results are summarized in the tables below:

	multiplications	additions
aliasing	0	N
modulation	2N	N
N/2-point complex DFT	NlogN - 2N	(3/2)NlogN - 2N
total	NlogN	(3/2)NlogN

Table 2. Cost of N-point complex DFT-odd operation.

	multiplications	additions
N/2-point complex DFT-odd	(N/2)logN - N/2	(3/4)NlogN - (3/4)N
spectral separation	N/2	N/2
spectral recombination	N	N
total	(N/2)logN + N	(3/4)NlogN + (3/4)N

Table 3. Cost of N-point real DFT-odd operation.

	multiplications	additions
even points	0	N/2

N-point real DFT-odd	(N/2)logN + N	(3/4)NlogN + (3/4)N
total	(N/2)logN + N	(3/4)NlogN + (5/4)N

Table 4. Cost of N-point real DFT given previous half-sized spectra.

Computation cost of no-latency method

For each block of the impulse response, we will determine the cost to compute one block of output samples (i.e. one block of yn) and divide by the blocksize to determine the cost per output sample. We will then sum this cost across all blocks of the filter response to obtain total cost per output sample. Using the direct form filter, an N-point convolution requires N^2 multiplications per N outputs. The block convolution method has three variations which depend on how we compute the spectrum of the input signal. Starting with no prior results, we need to do a real DFT to transform the input signal. For an N-point block convolution, the cost of a 2N-point real DFT plus the cost of an N-point block convolution operator is 3NlogN + 6N multiplications and (9/2)NlogN + 8N additions. If we have the two previous half-sized spectra, the cost is 2NlogN + 7N multiplications and 3NlogN + 7N additions per N outputs. If we have the input spectrum already computed, the cost is NlogN + 4N multiplications and (3/2)NlogN + 3N additions. These results are summarized in the table below:

	multiplications	additions
no prior results	3NlogN + 6N	(9/2)NlogN + 8N
given previous half-sized input spectra	2NlogN + 7N	3NlogN + 7N
given input spectrum	NlogN + 4N	(3/2)NlogN + 3N

Table 5. Cost of N-point block convolution.

The basic block size, N, is determined by the smallest block size for which block convolution is faster than direct form filtering. For an N-point block, direct form filtering requires N multiplications per output and block convolution requires 3logN + 6 multiplications (which assumes we have no prior results of the input spectrum). Therefore, in terms of real multiplications, block convolution will be more efficient starting at N = 32.

We have calculated the cost per output sample for the zero latency scheme introduced in figure 3 with N = 32. Figure 6 shows the number of multiplications required per output sample as a function of filter response size for various convolution algorithms. On one extreme is direct form filtering, which is a zero latency algorithm and has linearly increasing cost with filter size. On the other extreme is the very efficient large block convolution method whose cost grows very slowly with increasing filter size, but whose latency increases with increasing filter size. Bounded by these two is the cost for our zero latency block convolution algorithm, which begins asymptotic to the direct form curve and eventually becomes asymptotic in slope with the large block method. For example, a 512-point filter would require 512 multiplications per output sample for a direct form implementation, 155 multiplications for the zero latency algorithm, and 33 multiplications for a 23 millisecond latency large block algorithm. Also shown in the figure is the minimum cost solution according to figure 1. Note that the zero latency algorithm is not much more expensive than the minimum cost solution and has a similar growth curvature. If we don't reuse the input spectra as described earlier, the growth curve of the zero latency algorithm diverges from the minimum cost solution.

It is possible to trade off fixed latency for decreased cost. For instance, we can eliminate the direct form filter, and decompose the filter response into blocks of size N, N, 2N, 2N, 4N, 4N, etc. This algorithm will have a latency of 1.5 milliseconds for N = 32 (at a 44.1 kHz sampling rate). The cost for this algorithm is also shown in figure 6, it is simply the zero latency cost shifted down and to the left.

<how's about really big sizes? If the algorithm uses klogN multiplications per output sample, what is k?>

Multiprocessor considerations

Up till now, we have only considered a single processor architecture. We will now briefly discuss a multiprocessor implementation of the zero latency algorithm. Clearly, it makes sense to assign a different portion of the filter response to each processor. Thus, each processor needs access to the input signal, and each creates a convolution result that must be summed over all processors to generate the final result. Regardless of the architecture, this communication overhead should be slight, since each processor need receive and send only one sample per sampling period. A more serious communication constraint involves the reuse of input signal spectra, which is the source of significant optimization. In this case, the processor which is convolving with h_n must provide spectral results to the processor convolving with h_{n+1}. If possible, this transfer should happen instantaneously (i.e. through shared memory). Otherwise, the h_{n+1} processor must wait for the result to be transferred before it can use the result. If the size of h_n equals the size of h_{n+1}, then it probably makes sense to wait, since it will probably take less time to transfer the result between processors than to recompute it from scratch. If, however, the size of h_{n+1} is twice the size of h_n, then the h_{n+1} processor can start on the DFT-odd portion of the input spectrum computation while the h_n processor sends the half-sized results (even spectrum samples). One way to minimize the communication requirement is to have each processor compute contiguous blocks of the filter response. Thus, processor p_0

computes h_0 through h_a, processor p_1 computes h_{a+1} through h_b, processor p_2 computes h_{b+1} through h_c, etc. This scheme works particularly well if high bandwidth communication is available between processor p_n and p_{n+1} for transferring input spectra (i.e., a linear chain of processors). Of course, there will be a maximum blocksize which a processor can compute continuously. Once all smaller blocks of the filter response have been assigned to processors, all remaining processors should compute using this maximum blocksize.

Conclusions

We have described a method of convolution that has no input/output latency. The method is a hybrid approach combining direct form filtering and overlap-save block convolution. Because block transform techniques are used to render later portions of the filter response, the algorithm is significantly faster than direct form filtering, though slower than the highly efficient large block technique. Additional savings are achieved through reuse of input signal spectra, and fixed latency can be traded for computational efficiency. We have opted for a solution that keeps the processor evenly loaded over time, which performs close to the theoretical maximum. We have also described ways to implement this algorithm on a multi-processor architecture. One can imagine a device which could sample room impulse responses and then later render the room's reverberation in real-time by convolving an input signal with the measured room response. Using this zero-latency algorithm, such a device would be useful for many real-time applications.

The author would like to thank Keith Martin for his invaluable discussions and enthusiam for this work.

References

[1] T. G. Stockham, Jr., "High-speed convolution and correlation," in *Spring Joint Computer Conf., AFIPS Conf. Proc.*, vol. 28, pp. 229-233, 1966. Also in [9].

[2] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[3] C. S. Burrus, "Efficient Fourier transform and convolution algorithms," in *Advanced Topics in Signal Processing*, J. S. Lim and A. V. Oppenheim, Eds. Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[4] C. S. Burrus and T. W. Parks, *DFT/FFT and Convolution Algorithms*. New York: Wiley, 1984.

[5] M. Heideman and C. S. Burrus, "On the number of multiplications necessary to compute a length-2^n DFT," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-34, no. 1, February 1986.

[6] H. V. Sorenson, D. L. Jones, M. T. Heideman, and C. S. Burrus, "Real-valued fast Fourier transform algorithms," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, no. 6, June 1987.

[7] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965. Also in [9].

[8] L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[9] L. R. Rabiner and C. M. Rader, Eds., *Digital Signal Processing*, Selected Reprints, IEEE Press, New York, 1972.

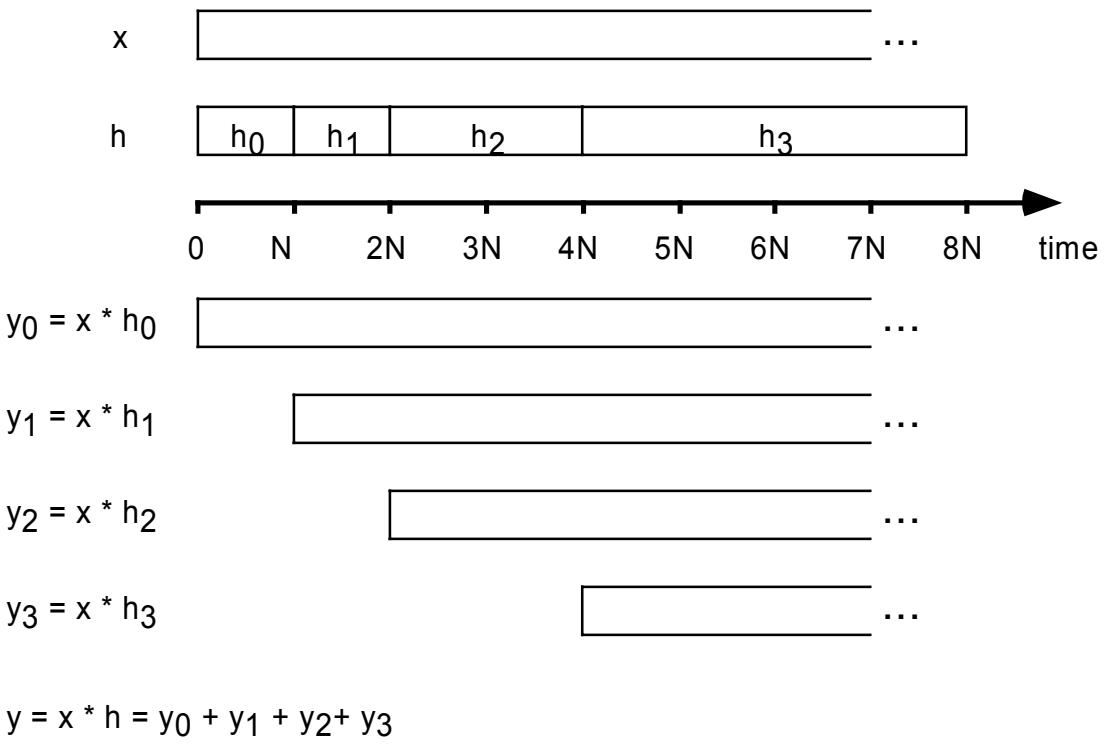


Figure 1a. Minimum cost decomposition of filter response.

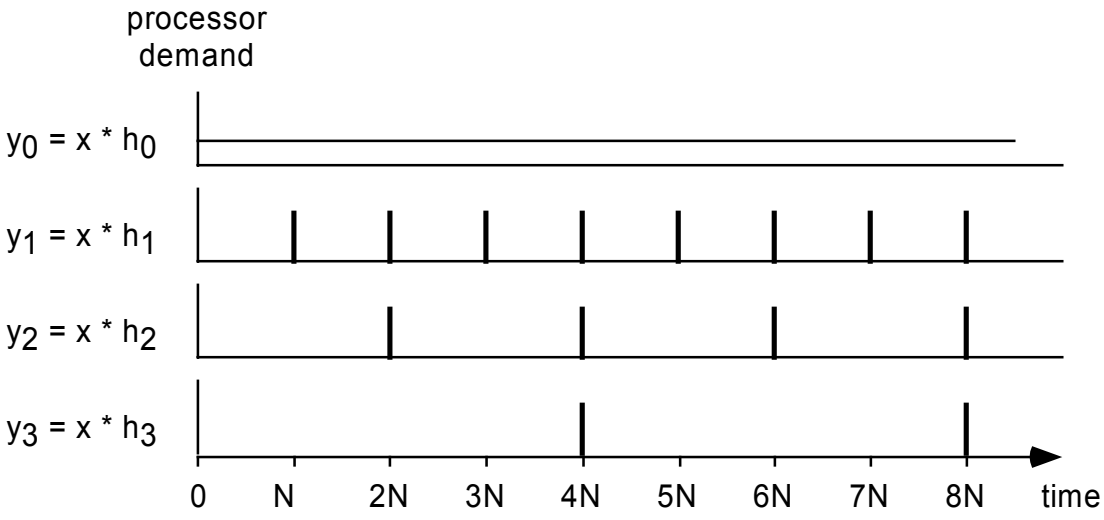


Figure 1b. Processor demand resulting from figure 1a.

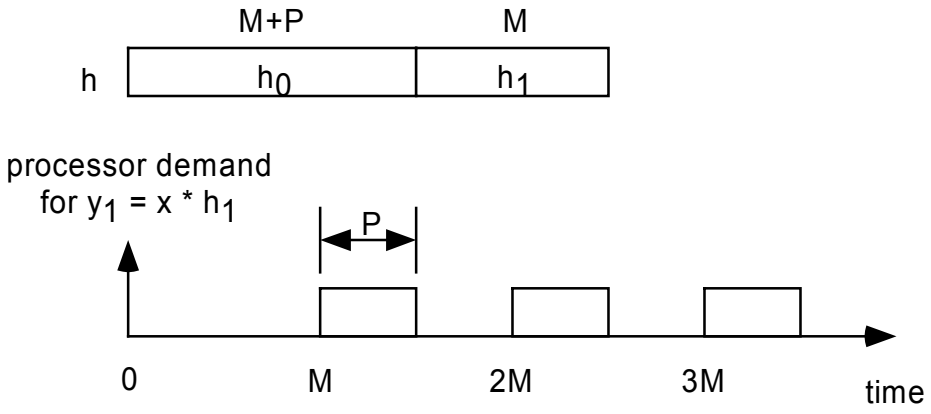


Figure 2. Non-uniform processor demand.

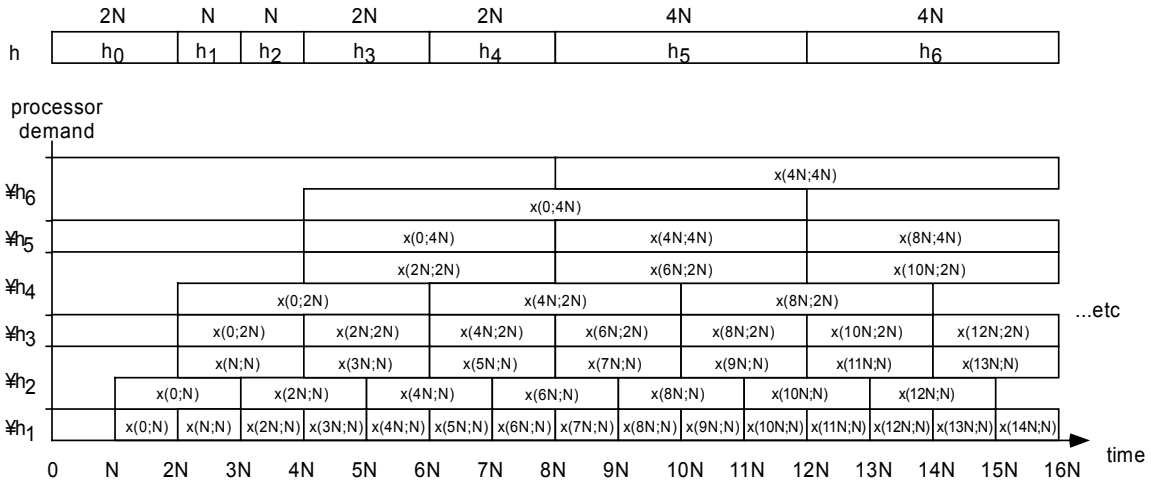


Figure 3. Uniform processor demand decomposition of filter response.

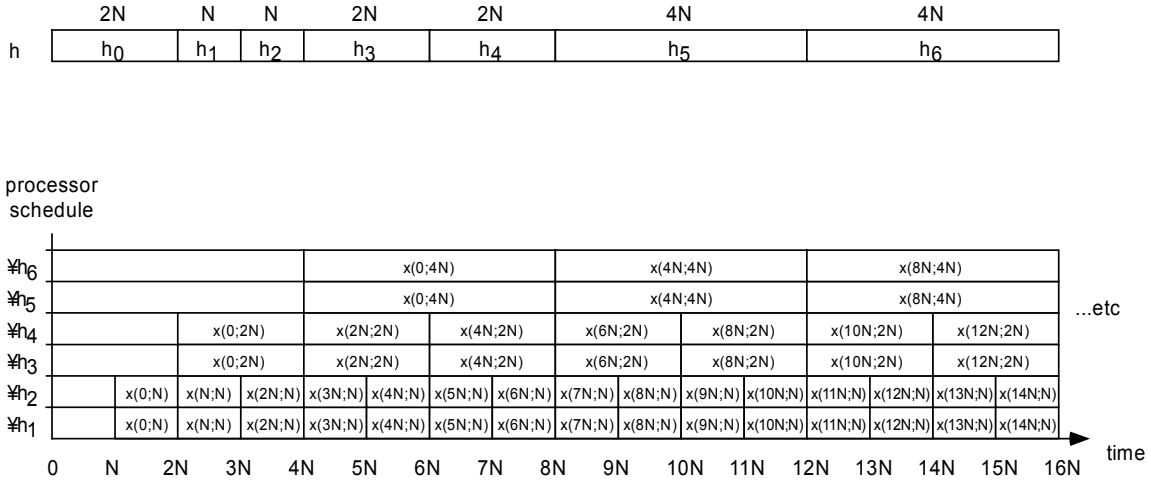
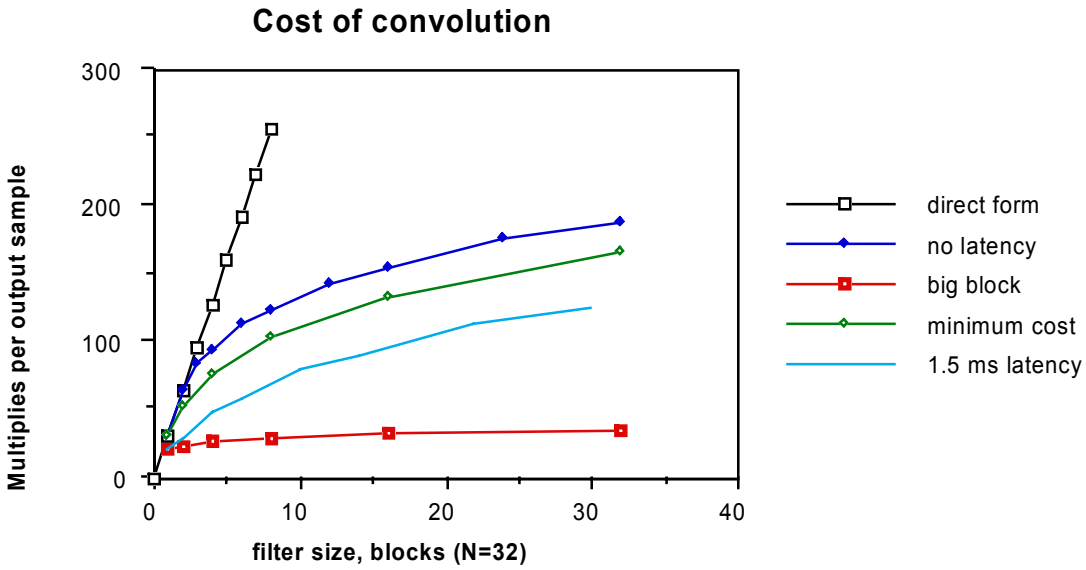


Figure 4. Processor scheduling.


Cost of convolution
Multiplies per output sample
300
200
100
0
0
10
20
30
40
filter size, blocks (N=32)
direct form
no latency
big block
minimum cost
1.5 ms latency