Efficient Convolution without Input–Output Delay*

WILLIAM G. GARDNER, *AES Member*

Perceptual Computing Section, MIT Media Lab, Cambridge, MA 02139, USA

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

0 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 impulse response is convolved with a source sound to simulate the room. Typical room responses can span several seconds, and thus a 2-s room response will require an 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 delay, because the processor must wait for an entire block of input samples to accumulate, and then perform significant calculations, before a block of output samples is generated. The processor calculates the correct output signal, but this signal is delayed due to internal buffering inherent in the block algorithm. If the processor is operating at maximum capacity, we can expect the input–output delay to be twice the length of the block size. (This constraint is discussed later.) Consequently, if we are convolving with a 2-s room impulse response, the input–output delay might be 4 s. This delay is prohibitive for most real-time performance or mixing applications.

There is a simple way to harness the efficiency of block transform methods without imposing any input–output delay. The linearity of convolution allows us to split the filter impulse response into blocks of different sizes, compute the convolution of each block with the input signal, and sum these results to obtain the output signal. Prior to summing, the results must be delayed according to the position of the block within the filter response. A block starting at sample M within the filter response will yield a result that must be delayed by M samples. Convolving with the first block of the filter response yields a result that is not delayed, and thus it must be computed using the direct-form FIR filter. Subsequent blocks of the filter response yield results that are delayed, so these may be computed using block FFT methods without violating the constraint of no input–output delay, provided we choose the block sizes correctly. If we want to use the largest possible block sizes without violating the delay constraint, we split the filter response into small blocks at the start of the response, with successively increasing block sizes at later times in the filter response.

This paper describes such an algorithm for doing efficient convolution without input–output delay. The method requires computing both a direct-form filter and sets of various sized block FFT convolutions. The design problem is how to split up the filter response into blocks, 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. The convolution algorithm can be optimized significantly by reusing previously computed spectra whenever possible. We will compare our hybrid method to both direct-form and standard large-block transform methods. As expected, our method is far more efficient

* Presented at the 97th Convention of the Audio Engineering Society, San Francisco, CA, 1994 November 10–13.

than direct-form implementation, but less efficient than large-block, large-delay methods. With our method, a small amount of fixed input–output delay can be traded for additional efficiency.

1 CONVOLUTION

The convolution operator $*$ is defined as follows:

$$y[n] = x[n] * h[n] \tag{1}$$

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \, . \tag{2}$$

The convolution summation in Eq. (2) can be easily implemented using the direct-form FIR filter, as shown in Fig. 1. This implementation requires N multiplies per output point.

A much more efficient convolution method uses the discrete Fourier transform (DFT) to perform the convolution in the frequency domain. An N-point block convolver using the overlap–save method is shown in Fig. 2. This has been drawn to look like a device that works one sample at a time; for each input sample shifted in at the top, an output sample is produced at the bottom. Internally the device computes N points of result every N sampling periods. The spectrum of the zero padded filter response can be calculated once before operation. During operation, samples are shifted into the input buffer. Every N samples, the forward DFT transform of the input samples, the spectral product with the filter spectrum, and the inverse DFT transform are calculated. Thus the overall input–output delay is N samples plus the time it takes to perform these calculations. This implementation is efficient because we can use the FFT algorithm to implement the DFT. The FFT block convolver requires $O(\log N)$ multiplies per output point.

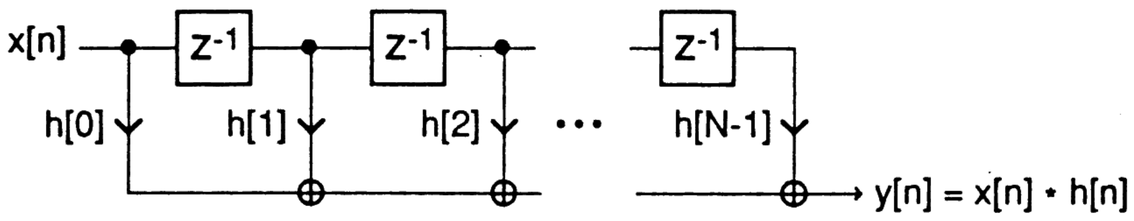


Fig. 1. Direct-form implementation of convolution.

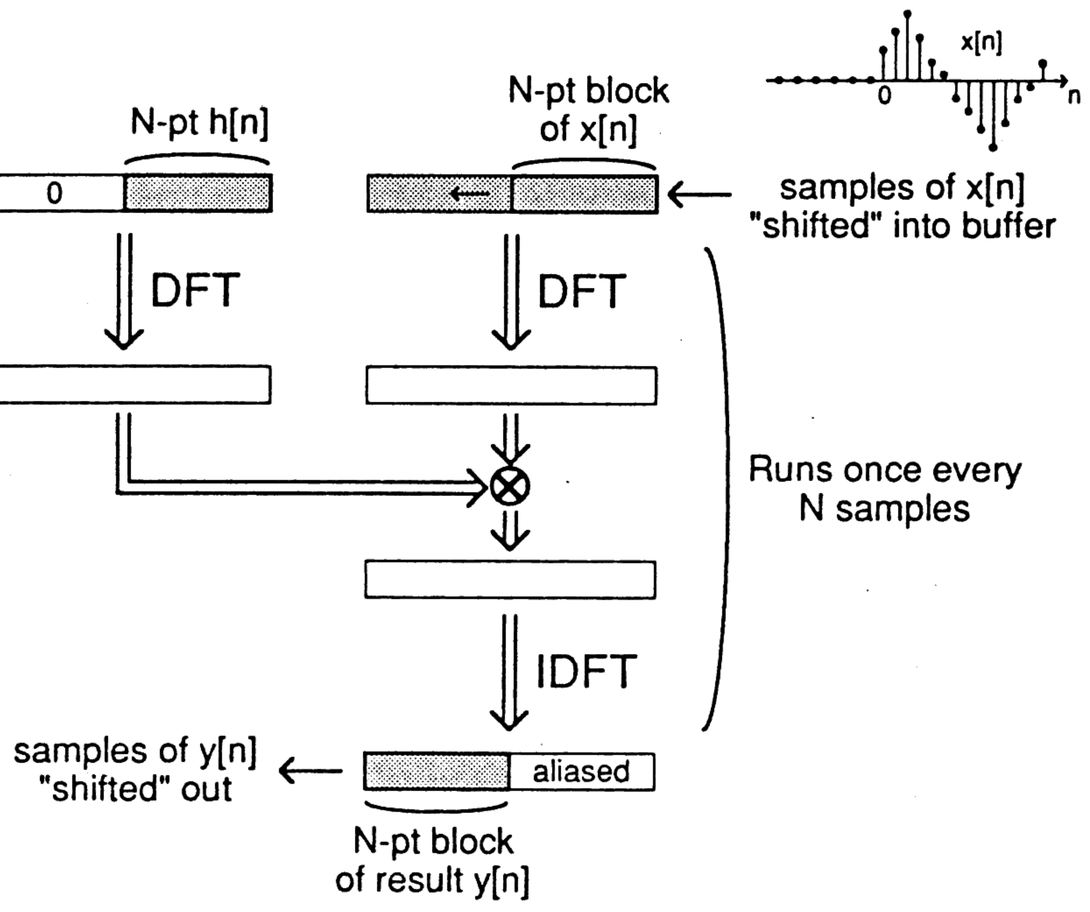


Fig. 2. N-point overlap–save block convolver.

1.1 Notation

We now introduce a notation to describe the operation of this block convolver,

$$h * x(n_0, N) \, . \tag{3}$$

This represents the N-point result of the convolution $y[n] = h[n] * x[n]$ starting at sample n_0, and we will use this notation to refer to the particular block operation that produced this N-point result. The output $y[n]$ can be produced by concatenating the results of many block convolution operations as follows:

$$y[n] = h * x(0, N) \mid h * x(N, N) \mid h * x(2N, N) \ldots \tag{4}$$

where $\mid$ represents concatenation.

2 ZERO-DELAY CONVOLUTION ALGORITHM

We will develop the zero-delay algorithm by first introducing a minimum-cost algorithm that is completely impractical. Then we will develop a practical algorithm that is nearly as efficient. We will constrain ourselves to using only radix-2 FFT algorithms [3], and thus all convolution block sizes will be powers of 2.

2.1 Minimum-Cost Algorithm

It is easy to divide the filter response into blocks such that we minimize the total computation cost per output point without violating the delay constraint. This is done by using the largest possible blocks to calculate the convolution. Fig. 3 shows a minimum-cost decomposition of the filter response.

The filter response h is $8N$ points long and is split into blocks h_0, h_1, h_2, and h_3 with N, N, $2N$, and $4N$ points, respectively. The first block, h_0, is N points long, and its result, y_0, is computed by direct-form filtering without

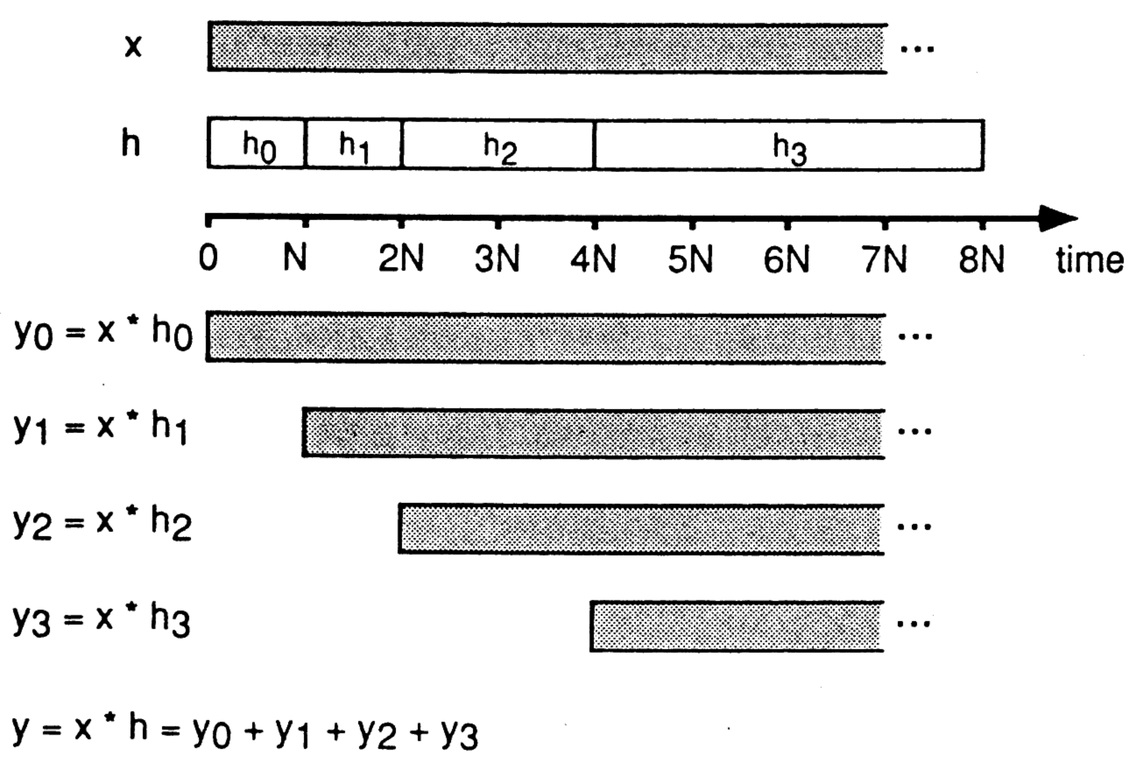


Fig. 3. Minimum-cost decomposition of filter response under constraint of no input–output delay.

delay. 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-point 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 $2N$-point 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 by adding blocks of size $8N$, $16N$, $32N$, and so on.

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 delay constraint. However, this algorithm is completely impractical because all the block computations must be done almost instantaneously. Each M-point block convolver must wait M sampling periods for the M-point block of input samples to be shifted in, and since the result cannot be delayed more than M samples, the computation must be complete before the next sampling period. If we had a processor that was fast enough to perform these calculations, we would be wasting its power, since most of its time would be spent waiting for input samples to be shifted into the input buffer. 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.

2.2 A Practical, Uniform 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 become nonuniform over time. Consider the M-point block h_1, which starts L points into the filter response, as shown in Fig. 4. The block convolution with h_1 is a repetitive computation which is done every M samples. The first block of the result $x * h_1$ must be computed by time L, but the computation cannot begin until time M. Thus the demand on the processor is nonuniform unless $L = 2M$. In other words, we allow the processor the same amount of time to do the block computation as it takes to wait for a block of input samples. With this constraint, each block of size M samples should start at least $2M$ samples into the filter response. This ensures that there is no time during which the processor is prevented from working on any particular computation.

3 IMPLEMENTATION

Fig. 5 illustrates how to decompose the filter response so that the processor demand stays constant. Fig. 5(a) shows a filter response of size $16N$. The first block, h_0, is of length $2N$, and its convolution result will be computed using a direct-form filter. The remaining blocks h_1 through h_6 have sizes N, N, $2N$, $4N$, and $4N$, respectively. With the exception of h_0, each block of size M begins at least $2M$ samples into the filter response. If h were longer, we would continue partitioning it using two blocks of size $8N$, followed by two blocks of size $16N$, and so on. As before, N is chosen to be the smallest block size at which block convolution is more efficient than the direct-form filter.

Fig. 5(b) shows the schedule for performing the various block convolution operations. Each horizontal strip shows the schedule of operations for one block of the

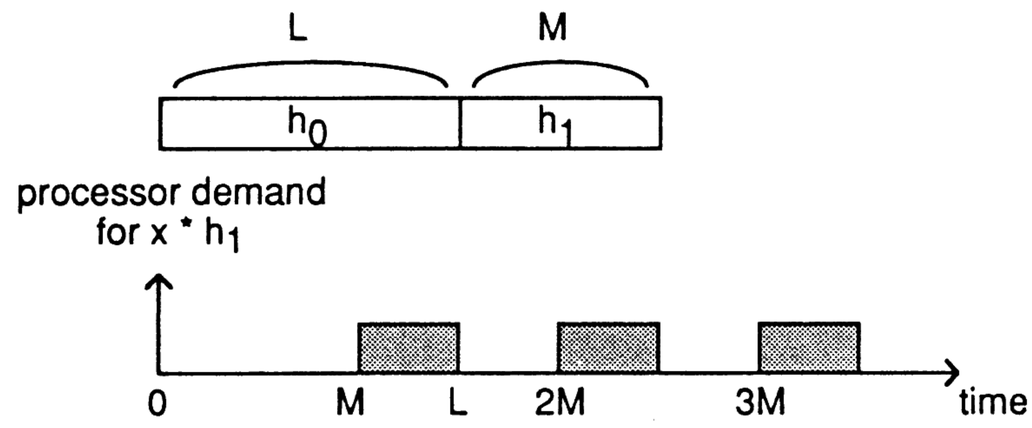


Fig. 4. Demand on processor is nonuniform over time unless blocks of size M start at least $2M$ samples into filter response.

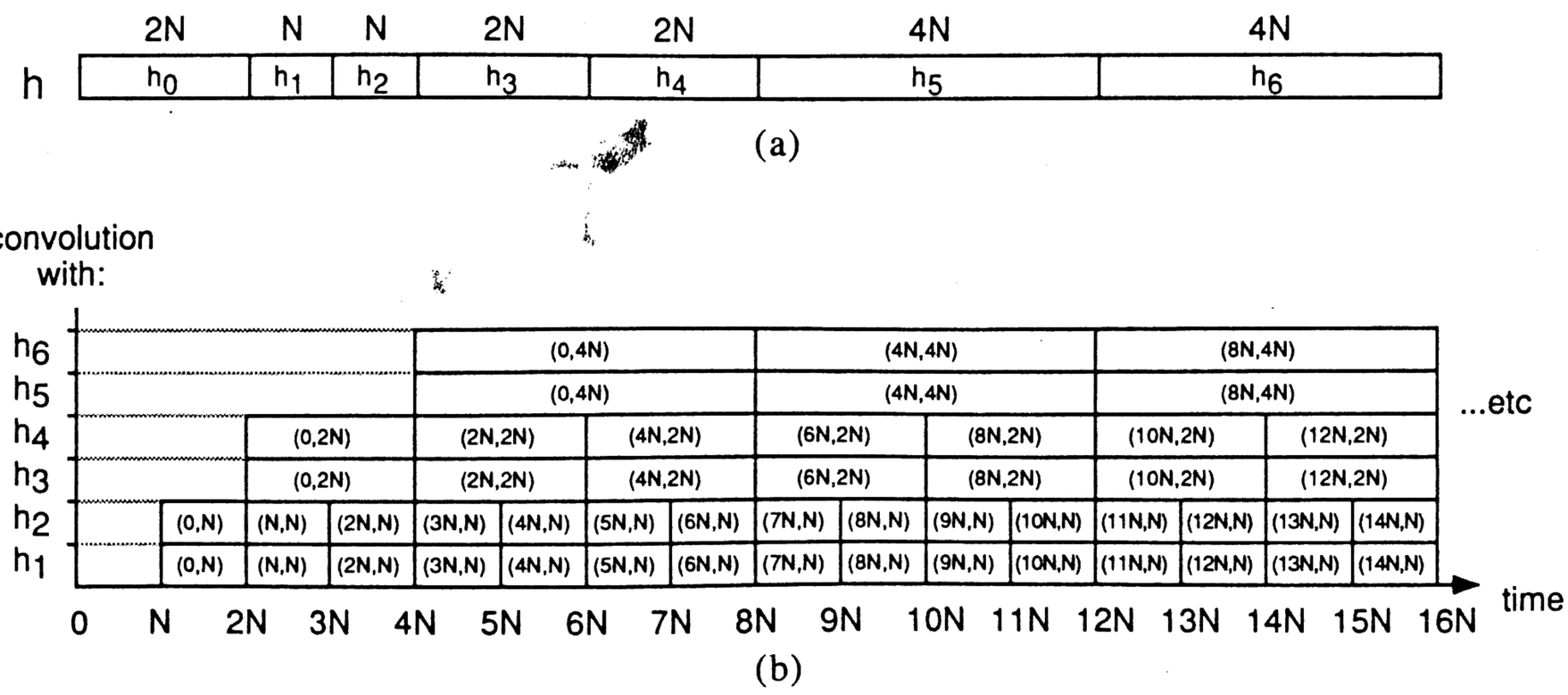


Fig. 5. (a) Practical decomposition of filter response. (b) Corresponding block convolution scheduling.

filter response using the notation introduced in Eq. (3). For instance, the convolution of the input with h_1 starts at time N with the computation of $x * h_1(0, N)$. This computation can begin no sooner than time N and must be complete by time $2N$. At time $2N$ we can start with the computation of $x * h_1(N, N)$. For the even-numbered blocks of h we have artifically imposed a deadline sooner than necessary. For instance, the computation of $x * h_2(0, N)$ can begin no sooner than time N, but does not have to be complete until time $3N$, although we are imposing a deadline of time $2N$. The processor is uniformly loaded over time with this schedule, except at the very start of the operation. Note that equal areas in the diagram do not necessarily correspond to equal amounts of computation; the diagram only shows the order of operations for each block convolver.

3.1 Process Scheduling

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 h_0 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 will have the highest priority. The $2N$-point tasks will have medium priority, and the $4N$-point tasks will have the lowest priority. So for this example we have three priority levels of tasks that must be completed. Table 1 shows what tasks are scheduled at times $t = 4N, 5N, 6N$, and $7N$. This scheduling follows directly from Fig. 5.

At time $t = 4N$ we will schedule tasks at all three priority levels. The highest priority task is to calculate $x * h_1(3N, N)$ and then calculate $x * h_2(3N, N)$. We want to do these sequentially, because we can use the intermediate results of the $x * h_1(3N, N)$ calculation to make the $x * h_2(3N, N)$ calculation faster. In particular, the same DFT of the input block is needed 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.

3.2 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. Tasks running for the first time are initiated via a jump to subroutine instruction, so that when the task completes, we return to the scheduler, and the next item on the list is executed. Every N samples the interrupt routine 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, and a sample interrupt has occurred. We service the interrupt, perform the direct-form FIR filter, and output the result sample by summing the separate convolution results. Now we would normally return from the interrupt to resume the current task, but because this is an N-sample boundary, there are higher priority tasks to be scheduled. Therefore we save the context of the current task, add the new tasks to the task list, and return to the process scheduler to sequentially execute the new tasks. For a typical DSP (such as the 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 current task's entry in the task list. After adding the new tasks to the task list, we 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 routine), 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 whether 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 just 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

Table 1. Task scheduling from Fig. 5.

Time $t = 4N$	Task
High priority:	$x * h_1(3N, N)$, $x * h_2(3N, N)$
Medium priority:	$x * h_3(2N, 2N)$, $x * h_4(2N, 2N)$
Low priority:	$x * h_5(0, 4N)$, $x * h_6(0, 4N)$
$t = 5N$	
High priority:	$x * h_1(4N, N)$, $x * h_2(4N, N)$
$t = 6N$	
High priority:	$x * h_1(5N, N)$, $x * h_2(5N, N)$
Medium priority:	$x * h_3(4N, 2N)$, $x * h_4(4N, 2N)$
$t = 7N$	
High priority:	$x * h_1(6N, N)$, $x * h_2(6N, N)$

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.

4 OPTIMIZATION

There are several ways the zero-delay convolution algorithm can be optimized.

1) Precalculate the spectra of all filter response blocks.

2) Optimize the basic block convolution operation by using real input FFTs, exploiting symmetry in the spectral product, and so on.

3) Reuse input block spectra directly whenever possible. This happens quite often in this algorithm. For instance, referring to Fig. 5, we see that every computation involving h_2 can use the input spectra already calculated for the h_1 computation. Therefore each h_2 block convolution operation only requires computing the spectral product and the inverse FFT, since the forward FFT of the input signal has already been calculated. This is true for all even-numbered blocks of h (except h_0, of course).

4) Calculate large input block spectra using the results of smaller input block calculations. This is a subtle but significant optimization, which is discussed in the next section.

4.1 Calculation of Input Spectrum from Half-Sized Results

We will consider the calculation of the forward DFT of the block of input samples, as shown in Fig. 2. For an M-point block convolution, the size of the actual DFT calculated is $2M$ points. For instance, consider the operation $x * h_3$ $(2N, 2N)$. This requires that we calculate the DFT of $4N$ input samples beginning at sample 0. Similarly, calculating $x * h_1$ (N, N) involves the $2N$-point DFT of input samples 0 through $2N - 1$, and calculating $x * h_1$ $(3N, N)$ involves the $2N$-point DFT of input samples $2N$ through $4N - 1$. Referring to Fig. 5, and remembering how processes are prioritized, we see that the two computations $x * h_1$ (N, N) and $x * h_1$ $(3N, N)$ will have completed before the computation $x * h_3$ $(2N, 2N)$ is started. Thus we already have the spectra of the first and second halves of the input block we are transforming. It is easy to see that this trivially yields the even points of the desired result spectrum by summing the half-sized spectra, and it is only necessary to calculate from scratch the odd points of the result. Calculating the odd points of a spectrum requires slightly more than half the total computation, so we have saved nearly half the computation associated with transforming blocks of input samples. This optimization is applicable to the block convolvers associated with odd-numbered blocks of h, except h_1, for which no half-sized results are available. The details of calculating the odd points of a spectrum using the "DFT-odd" operation are given in Appendix 2.

5 COMPARISON OF ALGORITHMS

We now compare the computational cost of the zero-delay algorithm with the direct FIR approach and the large-block, large-delay 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 [4]–[7]. A consideration of extensive optimization techniques is beyond the scope of this paper. Instead, we have prepared operation counts of the various computations assuming a reasonable amount of optimization, and we will simply list these in tables without explaining exactly how they were derived. The important factors are the proportionality constants of the most significant terms, and these agree with results from the literature.

5.1 Operation Cost

Unless explicitly stated otherwise, references to multiplications and additions will be assumed to be real, and all references to the log operation are base 2. We start with an N-point radix-2 complex DFT, which requires $(N/2) \log (N/2)$ complex multiplications and $N \log N$ complex additions. If complex multiplications are implemented using four multiplications and two additions (an alternative approach requires three multiplications and three additions [5]), this yields $2N \log N - 2N$ multiplications and $3N \log N - N$ additions for an N-point complex DFT. An N-point real DFT requires an $N/2$-point complex DFT plus some other operations (see Appendix 1). The summary of costs for the N-point real DFT is given in Table 2.

Using a similar method, but paying more attention to optimization details, Sorenson et al. [7] report $N \log N$ $^7/_2N - 6$ multiplications and $^3/_2N \log N - N/2 - 2$ additions. The proportionality constants of the significant terms agree with our results.

We now consider the cost of calculating an N-point real DFT given we already have the half-sized spectral results. These costs are summarized in Table 3. As discussed earlier, the even points of the result are obtained by adding the half-sized spectra, and the odd points of the result must be calculated from scratch. This algorithm is described in detail in Appendix 2.

The block convolution operation requires computing

Table 2. Computational cost of N-point real DFT.

	Multiplications	Additions
$N/2$-point complex DFT	$N \log N - 2N$	$^3/_2N \log N - 2N$
Spectral separation	0	N
Spectral recombination	$2N$	$2N$
Total	$N \log N$	$^3/_2N \log N + N$

the forward transform, the spectral product, and the inverse transform. The cost of the inverse transform is assumed to be the same as the real input FFT shown in Table 2. There are three possibilities for the forward transform: 1) we have no prior result and must calculate it from scratch (applicable to the h_1 convolver); 2) we use previously calculated half-sized spectra (odd-numbered convolvers); and 3) we have already calculated the forward transform and reuse it directly (even-numbered convolvers). With these conditions, the costs of the N-point block convolution are summarized in Table 4.

5.2 Cost of Zero-Delay Algorithm

For each block of the filter response, we determine the cost to compute one block of partial output samples (that is, one block of y_n) and divide by the block size to determine the cost per partial output sample. We then sum this cost across all blocks of the filter response to obtain the total cost per output sample.

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 $3 \log N + 6$ 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-delay scheme with $N = 32$. Fig. 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-delay algorithm but 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 input–output delay increases with increasing filter size. Bounded by these two is the cost for our zero-delay hybrid block convolution algorithm. Also shown in the figure is the minimum-cost, zero-delay solution according to the filter decomposition shown in Fig. 3. Note that the practical zero-delay algorithm is not much more expensive than the minimum-cost zero-delay algorithm and has a similar growth curvature.

It is possible to trade off fixed input–output delay for decreased cost. For instance, we can eliminate the direct-form filter and decompose the filter response into blocks of size N, N, $2N$, $4N$, $4N$, and so on. This algorithm will have an input–output delay of 1.5 ms for $N = 32$ (at a 44.1-kHz sampling rate). The cost for this algorithm is also shown in Fig. 6. It is simply the zero-delay cost shifted down and to the left.

The cost of the zero-delay algorithm is approximately $34 \log_1(n) - 151$ multiplies per output sample for filter size n. Thus a filter of size 128K samples will require approximately 427 multiplies per output sample. This compares favorably with the 128K multiplies per output sample required to implement a direct-form filter. A single large-block convolution would require 68 multiplies per output sample, but would have an input–output delay of 6 s.

6 MULTIPROCESSOR CONSIDERATIONS

Up till now we have only considered a single processor architecture. We will now briefly discuss a multiprocessor implementation of the zero-delay 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 must receive and send only one sample per sampling period. A more serious communication constraint involves the reuse of input signal spectra, which are the source of significant optimization. In this case, the processor that is convolving with h_n must provide spectral results to the processor convolving with h_{n+1}. If possible, this transfer should happen instantaneously (that is, through shared memory). Otherwise the h_{n+1}

Table 3. Cost of N-point real DFT given previous half-sized spectra.

	Multiplications	Additions
Even points	0	$N/2$
N-point real DFT-odd	$(N/2) \log N + N$	$\frac{3}{4}N \log N + \frac{3}{4}N$
Total	$(N/2) \log N + N$	$\frac{3}{4}N \log N + \frac{5}{4}N$

Table 4. Cost of N-point block convolution.

	Multiplications	Additions
No prior results	$3N \log N + 6N$	$\frac{9}{2}N \log N + 8N$
Given previous half-sized input spectra	$2N \log N + 7N$	$3N \log N + 7N$
Given input spectrum	$N \log N + 4N$	$\frac{3}{2}N \log N + 3N$

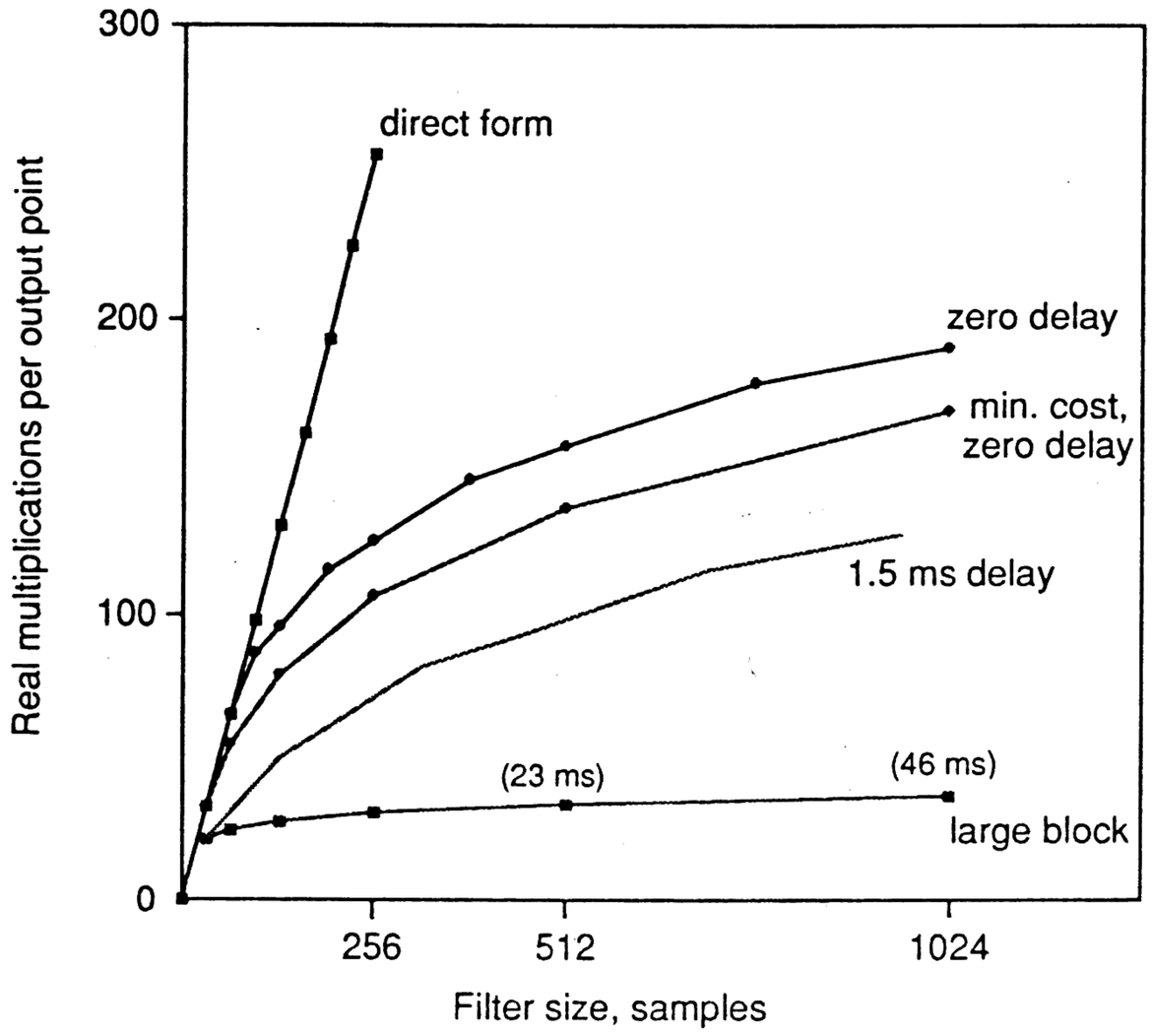


Fig. 6. Comparison of different algorithms.

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 compute 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, and so on. This scheme works particularly well if high-bandwidth communication is available between processors p_n and p_{n+1} for transferring input spectra (that is, a linear chain of processors). Of course, there will be a maximum block size that 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 block size.

7 CONCLUSIONS

We have described a method of efficient convolution that has no input–output delay. 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 very efficient large-block transform technique. Additional savings are achieved through the reuse of input signal spectra, and fixed delay 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 multiprocessor architecture.

Devices that can render room reverberation in real time by convolving an input signal with a measured room response are already on the market. It is anticipated that zero-delay convolution algorithms will play a role in the success of these devices for real-time applications.

The author would like to thank Keith Martin, Paul Beckman, and Dan Ellis for their help in producing this paper.

8 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); reprinted in *Digital Signal Processing, Selected Reprints*, L. R. Rabiner and C. M. Rader, Eds. (IEEE Press, New York, 1972).

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

[3] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathe. of Comput.*, vol. 19, pp. 297–301 (1965); reprinted in *Digital Signal Processing, Selected Reprints*, L. R. Rabiner and C. M. Rader, Eds. (IEEE Press, New York, 1972).

[4] C. S. Burrus, "Efficient Fourier Transform and Convolution Algorithms," in *Advanced Topics in Signal Processing*, J. S. Lim and A. V. Oppenheim, Eds. (Prentice-Hall, Englewood Cliffs, NJ, 1988).

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

[6] M. Heideman and C. S. Burrus, "On the Number of Multiplications Necessary to Compute a Length-2^n DFT," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-34 (1986 Feb.).

[7] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus, "Real-Valued Fast Fourier Transform Algorithms," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. ASSP-35 (1987 June).

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

APPENDIX 1
THE REAL INPUT DFT

It is well known that an N-point real input DFT can be computed using an $N/2$-point complex DFT [7]. We will quickly go through the derivation so we can refer to it later in Appendix 2. 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 input sequences are the even and odd samples of an N-point real sequence, then the two $N/2$-point 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 for two N-point sequences 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 $G_{OR}[k]$, $G_{ER}[k]$, and $G_{OI}[k]$, 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

$$G_{OR}[k] = \frac{1}{2}\{G_R[k] - G_R[N-k]\} \tag{5}$$

$$G_{ER}[k] = \frac{1}{2}\{G_R[k] - G_R[N-k]\} \tag{6}$$

$$G_{OI}[k] = \frac{1}{2}\{G_I[k] - G_I[N-k]\} \quad (7)$$

$$G_{EI}[k] = \frac{1}{2}\{G_I[k] - G_I[N-k]\} \quad (8)$$

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

$$X_1[k] = G_{ER}[k] + jG_{OI}[k] \quad (9)$$

$$X_2[k] = G_{EI}[k] + jG_{OR}[k] \quad (10)$$

This shows how we would perform two independent N-point real DFTs using a single N-point complex DFT. A similar strategy applies for computing two $N/2$-point real DFTs using a single $N/2$-point complex DFT.

The DFT $X[k]$ of an N-point sequence $x[n]$ is defined as follows:

$$\text{DFT}\{x[n]\} = X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk} \quad (11)$$

where

$$W_N = e^{-j2\pi/N}. \quad (12)$$

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

$$X[k] = \sum_{n=0}^{(N/2)-1} x[2n]\, W_N^{2nk} + \sum_{n=0}^{(N/2)-1} x[2n+1]\, W_N^{(2n+1)k} \quad (13)$$

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

This is the formulation of the decimation-in-time FFT decomposition [2]. 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.

APPENDIX 2
THE DFT-ODD OPERATION

We will examine the method of computing 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 $0 \leq n < N/2$. All the needed relationships fall out of the derivation for the decimation-in-frequency FFT [8]. The N-point DFT of $x[n]$ is

$$\text{DFT}\{x[n]\} = X[k] = \sum_{n=0}^{N} x[n]\, W_N^{nk} \quad (15)$$

$$= \sum_{n=0}^{N/2-1} x_1[n]\, W_N^{nk} + \sum_{n=0}^{N/2-1} x_2[n]\, W_N^{(n+N/2)k} \quad (16)$$

$$= \sum_{n=0}^{N/2-1} \{x_1[n] + e^{-j\pi k} x_2[n]\}\, W_N^{nk}. \quad (17)$$

Let us consider the even points of the spectrum $X[k]$,

$$X[k = 2r] = \sum_{n=0}^{N/2-1} \{x_1[n] + e^{-j\pi 2r} x_2[n]\}\, W_N^{n2r} \quad (18)$$

$$= \sum_{n=0}^{N/2-1} \{x_1[n] + x_2[n]\}\, W_{N/2}^{nr} \quad (19)$$

$$= \sum_{n=0}^{N/2-1} x_1[n]\, W_{N/2}^{nr} + \sum_{n=0}^{N/2-1} x_2[n]\, W_{N/2}^{nr} \quad (20)$$

$$= X_1[r] + X_2[r]. \quad (21)$$

Eq. (21) shows that the even points of $X[k]$ can be determined by adding the spectra of x_1 and x_2. Let us consider the odd points of $X[k]$,

$$X[k = 2r+1] = \sum_{n=0}^{N/2-1} \{x_1[n] + e^{-j\pi(2r+1)} x_2[n]\}\, W_N^{n(2r+1)} \quad (22)$$

$$= \sum_{n=0}^{N/2-1} \{x_1[n] - x_2[n]\}\, W_N^n W_N^{n2r} \quad (23)$$

$$= \sum_{n=0}^{N/2-1} \{x_1[n] - x_2[n]\}\, W_N^n)\, W_{N/2}^{nr}. \quad (24)$$

Eq. (24) shows that the odd points of $X[k]$ can be determined by prealiasing and modulating $x[n]$ and then performing an $N/2$-point complex DFT. We will show how the calculation of Eq. (24) 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 us redefine Eq. (24) and call it the "DFT-odd" operation,

$$\text{DFT-odd}\{x[n]\} = X[k = 2r+1] = X'[r] = \sum_{n=0}^{N-1} x[n]\, W_N^{n(2r+1)} \tag{25}$$

for $0 \leqslant r < N/2$. If $x[n]$ is real, than $X'[r]$ will be symmetric,

$$X'[r] = X'^*\,[(N/2) - r - 1]\,. \tag{26}$$

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

$$X'[r] = -X'^*\,[(N/2) - r - 1]\,. \tag{27}$$

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.

$$X'[r] = \sum_{n=0}^{N-1} x[n]\, W_N^{n(2r+1)} \tag{34}$$

$$= \sum_{n=0}^{(N/2)-1} x[2n]\, W_N^{2n(2r+1)} + \sum_{n=0}^{(N/2)-1} x[2n+1]\, W_N^{(2n+1)(2r+1)} \tag{35}$$

$$= \sum_{n=0}^{(N/2)-1} x_1[n]\, W_{N/2}^{n(2r+1)} + W_N^{2r+1} \sum_{n=0}^{(N/2)-1} x_2[n]\, W_{N/2}^{n(2r+1)}\,. \tag{36}$$

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 $X_1'[r]$ and $X_2'[r]$, respectively. Let $g[n]$ be the complex sequence $x_1[n] + jx_2[n]$ with corresponding DFT-odd $G'[r] = G_R' + jG_R'[r]$, 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

$$G_{OR}'[r] = \frac{1}{2}\{G_R'[r] - G_R'[(N/2) - r - 1]\} \tag{28}$$

$$G_{ER}'[r] = \frac{1}{2}\{G_R'[r] - G_R'[(N/2) - r - 1]\} \tag{29}$$

$$G_{OI}'[r] = \frac{1}{2}\{G_I'[r] - G_I'[(N/2) - r - 1]\} \tag{30}$$

$$G_{EI}'[r] = \frac{1}{2}\{G_I'[r] - G_I'[(N/2) - r - 1]\}\,. \tag{31}$$

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

$$X_1' = G_{ER}'[r] + jG_{OI}'[r] \tag{32}$$

$$X_2' = G_{ER}'[r] + jG_{OR}'[r]\,. \tag{33}$$

Let us return to the DFT-odd formulation of Eq. (25) 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 $X'[r]$:

The summations in Eq. (36) 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 Eq. (24), which only requires doing an $N/4$-point complex DFT. We then perform a spectral separation according to Eqs. (28)–(33), and finally we perform the spectral recombination using a set of butterflies according to Eq. (36).

Cost estimates for the complex and real DFT-odd operations are given in Tables 5 and 6.

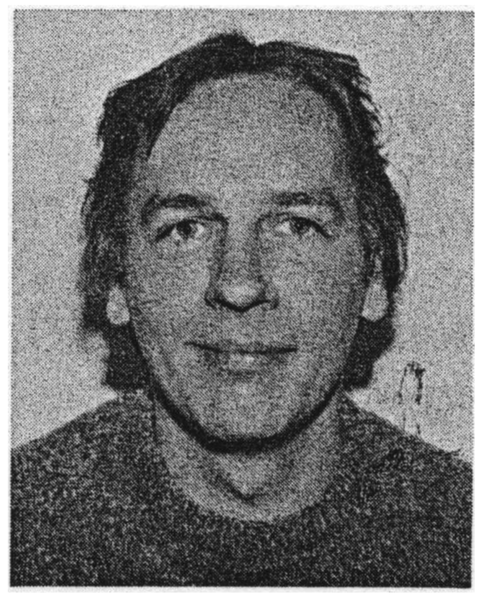
Table 5. Cost of N-point complex DFT-odd operation.

	Multiplications	Additions
Aliasing	0	N
Modulation	$2N$	N
$N/2$-point complex DFT	$N \log N - 2N$	$3/2N \log N - 2N$
Total	$N \log N$	$3/2N \log N$

Table 6. Cost of N-point real DFT-odd operation.

	Multiplications	Additions
$N/2$-point complex DFT-odd	$(N/2) \log N - N/2$	$3/4N \log N - 3/4N$
Spectral separation	$N/2$	$N/2$
Spectral recombination	N	N
Total	$(N/2) \log N + N$	$3/4N \log N + 3/4N$

THE AUTHOR



Bill Gardner was born in 1960 in Meriden, CT, and grew up in the Boston, MA, area. He received a bachelor's degree in computer science from MIT in 1982 and shortly thereafter joined Kurzweil Music Systems as a software engineer. For the next seven years, he helped develop software and signal processing algorithms for Kurzweil synthesizers. He left Kurzweil in 1990 to enter graduate school at the MIT Media Lab, where he is currently pursuing a Ph.D. as a Motorola fellow. During summers, he works at Lexicon developing audio effects algorithms. His current interests are reverberation, spatial audio, and real-time signal processing. Mr. Gardner is a member of the Audio Engineering Society and the Acoustical Society of America.