THE USE OF INTERPOLATING MEMORIES FOR MUSIC PROCESSING AT MICROCOMPUTER

ANDREW HOSTIEL

The Polytechnic Institute of New York
333 Jay Street
Brooklyn, New York 11201

ABSTRACT

A multimicroprocessor system designed for music synthesis has been enhanced by integrating the logic required for interpolation with random-access memory. The memory unit will hold values of functions used in music generation. On each read reference to the memory unit, an interpolated function value is obtained. The design was intended to speed up the process of interpolation, but it has been found useful for avoiding multiplication altogether in many music-processing algorithms. The design considerations of such a device, called an interpolating memory unit, are discussed. An example is presented of the use of the interpolating memory to speed up music processing.

1 INTRODUCTION

A low-cost multimicroprocessor system has been designed and implemented for real-time music synthesis. In the design, the high computation rates required for real-time processing are attained by allowing a high degree of parallelism through simple, inexpensive microprocessor boards. The microprocessors each have a private memory, and communicate with each other through a central shared memory. The prototype version of the system, described in Hostiel (1981), was implemented with four microprocessors. An eight-processor version is now being assembled.

Analysis of various music-generation algorithms and experimentation with different software organizations has made it apparent that techniques should be considered to speed up the individual microprocessors, instead of simply adding more processors. One consideration is the increase in overhead required for controlling a large number of processors, even when they are performing straightforward computational tasks. A second is that the central memory will become a system bottleneck. This is a concern especially for computations based on functions expressed in tabular form, which require that large tables be transmitted frequently among the microprocessors.

The need for speeding up the microprocessors can be seen in the elementary music processing step of reading and interpolating values from a waveform table. If a simple linear interpolation is used, one multiplication is required. The multiplication may take fourteen microseconds (as it does on the Intel 80186 with a 12-MHz clock), which leaves no time for any other processing during a sample period. The design of a fast arithmetical unit in semi-custom logic, as shown in Hostiel (1984), is an effective solution, but it is not inexpensive.

The solution adopted in the multimicroprocessor system is to move the interpolation step into the logic associated with a
memory read. This hardware unit is designated an interpolating memory. This solves several problems at once. First, the multiplication in the interpolation is reduced to a few bits and is implemented in combinational logic requiring little more time than the memory reference. Second, function tables may be reduced in size with no loss in precision. Third, the quick evaluation of the exponential and logarithm functions in the interpolating memory enables the multiply operation to be avoided altogether in many music processing algorithms.

In the following sections, we discuss the interpolating memory concept in general and present several different design options, and the implemented version. We show the relationship between the size of the interpolating memory and the number of bits involved in the interpolation arithmetic. The implementation of the exponential function is an example. We then show an example of eliminating multiplication through the use of the exponential and logarithm functions.

THE INTERPOLATING MEMORY DESIGN

Suppose \( f(x) \) is a function to be evaluated at \( 2^n \) points. Let \( x \) be the \( k \)-bit value of the argument. An interpolating memory is a memory unit that holds fewer than \( 2^n \) values of \( f(x) \), yet provides \( 2^n \) function values. We will assume the interpolating memory holds \( 2^n \) values of \( f(x) \), for \( k \leq n \), possibly along with other data related to \( f(x) \). The values of \( f(x) \) not held in the interpolating memory are determined through hardware interpolation on each memory reference.

A general outline of an interpolating memory is shown in Figure 1. The location of the binary point in \( x \) and \( f(x) \) is irrelevant to the hardware. To simplify the notation, \( x \) will be assumed to be in the form \( x_1 x_2 \), where \( x_1 \) is a \( k \)-bit value used to address the data stored in the interpolating memory, and \( x_2 \) is an \( n \)-bit value. We will call the \( k \) bits of \( x_1 \) the address bits, and the \( n \) bits of \( x_2 \) the interpolating bits.

One simple form of an interpolating memory might make two references to the stored data, \( f(x_1) \) and \( f(x_1+1) \), and then evaluate \( f(x) \) by the formula:

\[
f(x_2, x_2') = f(x_1) \cdot (1-x_2') + f(x_1+1) \cdot x_2'.
\]

This approach would require timing circuits and latches to effect the two memory references, and sacrifice the speed required in the function evaluation.

A more feasible implementation, shown in Figure 2, would employ a dual-port memory unit. In this case, the two memory references can be made almost simultaneously, with no need for timing circuitry. Two multiply units are required to complete the interpolation. However, the multiplications are by \( n \) bits, which might be only a small part of the total \( n \)-bit argument.

The implemented interpolating memory is shown in Figure 3. The value \( \delta y = f(x_1+1) - f(x_1) \) is stored in and read from a memory unit in parallel with \( f(x_1) \).
Then \( f(x) \) is evaluated from

\[
f(x_1, x_2) = f(x_1) + x_2 \Delta f(x_1)
\]

In this design, we have eliminated the dual-port RAM and one multiply unit.

The amount of stored data has increased, but by a factor of somewhat less than two. For most functions evaluated over a finite range, as the number of stored data values increases, the size of each of the increments \( \Delta f(x) \) will decrease proportionately. The maximum increment value may be expressed in only a fraction of the number of bits required for \( f(x) \).

2. PRECISION OF THE INTERPOLATION

The choice of the value of \( n \) will depend on the quality of the interpolation, which will be determined by the function \( f(x) \).

The characteristics of three important functions have been investigated in Noetzli & Wung (1985). These functions are the (base=2) logarithms, (base=2) exponential, and sine. Hereinafter, \( \log \) and \( \exp \) will refer to the base=2 functions.

For example, consider the evaluation of \( f(x) = \exp(x) \). If \( x_1 < x < x_2 \), then

\[
\exp(x) = \exp(x_1) \cdot \exp(0, x_2).
\]

The multiplication \( \exp(x_1) \) is simply a shift of the result \( \exp(0, x_2) \) by \( x_1 \) bit positions. The shifting operation can be accomplished by one quick step if it is implemented in hardware. Therefore, for the evaluation of \( f(x) = \exp(x) \), it is sufficient to be able to evaluate \( \exp(0, x_2) \) for \( 0 < x < 1 \). A similar result holds for the \( \log \) function. The sine function is periodic and in fact requires evaluation only over a quarter cycle.

Now consider the case of a linear approximation to \( \exp(x) \), with only one approximating segment over the entire interval. I.e., \( k=0 \) and there is only one value stored in the interpolating memory. This case is shown in Figure 4a. The maximum error is 0.0861. Since the function value is known to be between one and two, the accuracy of the approximation corresponds to a precision of between three and four bits.

Figure 4b shows the case of two approximating segments \( k=1 \). Here, the maximum error is reduced to 0.0261, which corresponds to a precision of more than five bits. In Figure 4c, four approximating segments are used \( k=2 \). The maximum error is reduced to 0.0041, or a precision of nearly seven bits. As the computation of the maximum error is continued for higher values of \( k \), the error continues to be reduced by a factor which is nearly constant. In this case, the preci-
The error in the result increases by 1.75 bits for each interpolating memory address bit.

In Moustel & Wong (1993), it is shown that $\log$, $\log$, and $\sin$ all have the characteristic of yielding about two bits of precision for each address bit, when a linear interpolation is used. Furthermore, when second degree interpolation is used, they yield three bits of precision for each address bit. With a third degree interpolation, four bits of precision are obtained for each address bit.

4 CHOICE OF INTERPOLATION PARAMETERS

For most functions used in signal processing, it is generally not useful to have function values expressed to a precision that is either greater or less than that of the argument.

Assume that both $z$ and $f(x)$ are to be expressed in $m$ bits, and the interpolation provides at least $1.5$ bits of precision for each of the $k$ address bits. Assume further that a linear segment over the entire range provides a precision of at least two bits. (These assumptions are quite conservative for the class of functions of interest.) Then for sufficient precision of the result,

$$2 + 1.5k = 2 + 1.5(m-n) > n$$

or,

$$n < \frac{m + 4}{3}$$

For example, a result with a precision of $10$ bits could be obtained with as few as ten bits for memory address, and six bits of interpolation. This is the configuration that was implemented. However, the complexity of the multiply unit and the total memory requirement must be considered in the interpolating memory design.

5 MEMORY REQUIREMENTS

The choice of the optimal value of the number of interpolating bits $n$ should take
into consideration the cost of memory and the cost and speed of the multiplication used in the interpolation. A smaller value of \( n \) results in not only fewer stages of multiplication, but smaller values of \( \Delta \), thereby reducing both the multiplication time and the word size of the memory units containing this data.

An upper bound on the value of \( \Delta \) for \( \exp(x) \) is the maximum slope in the range \( x = 1 \). This occurs at \( x = 1 \) and has value \( \ln(2) = 1.4427 \). The value of the upper bound of \( \Delta \) is divided by two for each of the \( k \) bits of the memory address. If \( k \) bits are used to express values in the range of zero to one, then the maximum value of \( \Delta \) is expressed in \( n-1-k \) bits.

For example, if \( n=16 \), then as long as \( k \leq 8 \), the values of \( \Delta \) can be contained in a memory unit with an \( 8 \)-bit word.

### 6 Elimination of Multiplication

The interpolating memory was designed to speed the interpolation of table-specified functions. But an additional benefit has been obtained from the computation speedup made possible by the exponential and logarithmic transformations. From

\[
x^y = \exp(\log x + \log y)
\]

it is obvious that a multiplication can be replaced by an addition and a table lookup. In the Intel 8086 processor, a multiplication takes about 160 clock cycles. The addition and table lookups take about 10 clock cycles each. Therefore, implementing multiplication by this straightforward log-exp transformation results in a speedup factor of approximately 3.5.

Greater efficiencies are possible by maintaining in logarithmic form the parameters and variables used in signal processing algorithms. Consider, for example, the computation of a \( p \)-th order recursive (all-pole) filter, as used in the LPC technique. (See Figure 5a.) The LPC filter can be expressed in the form of the filter of Figure 5b, which, for general values of the coefficients \( b_i \), also represents a more general (i.e., post-zero) filter. The output of the filter of Figure 4b can be expressed

\[
s(n) = a_1s(n-1) + a_2s(n-2) + \cdots + a_ps(n-p).
\]

To evaluate \( s(n) \), the required constants may be expressed in logarithmic form, as \( \log(a_i) \) and \( K_i = \log(b_i) \), for \( i=1,\ldots,p \). Also, let \( \Sigma_i = n \log(b_i) \). At each sample point,

\[
\Sigma_{i,n} = \Sigma_{i,n-1} + \log(2)
\]

is computed. Then,

\[
s(n) = \exp(K_1, n) \cdot \exp(K_2, n) \cdots \exp(K_p, n).
\]

Thus, each of the \( p \) multiplications required of the general filter are replaced by one addition and one table lookup. The elimination of multiplication in the implementation of the algorithm results in a speedup factor of about seven.

Comparable results have been found for other basic signal processing algorithms, such as the FFT. No precise measurements
have been made for the overall speedup
tained by the interpolating memory, but
the real-time capability of the multifur-
croprocessor system seems to have been at
least doubled.

REFERENCES

BRUNNER, T. A. & BECKER, J. C. (1975)
multiplication using logarithms implemented
with real-only memory. IEEE Transactions
on Computers, vol c-24, pp. 761-765, August
1975.

implementation. Proceedings of the 1984
Computer Music Conference, Paris, October
1984.

HOETZEL, A. S. (1982) A multiticrocom-
puter system for real-time music synthesis: de-
design and implementation. proceedings of
the 1982 Computer Music Conference, Venice,
October 1982.

Function evaluation by table lookup with
hardware interpolation, with applications.
Computer science Report no. 101, Department
of Computer Science, PolyTechnic Institute
of New York, Brooklyn, N.Y.