Characterizing and Enhancing the Performance of Sound Synthesis Applications on Pentium III

Course Project Final Report
CMU CS 740: Computer Architecture, Fall 2002
Prof. Seth Goldstein
Ning Hu (ninghu@cs.cmu.edu) and Vahe Poladian (poladian@cs.cmu.edu)

Project web page: http://www.cs.cmu.edu/~poladian/arch/project

Abstract

Computationally intensive applications such as sound synthesis, animation, scientific computation, can take advantage of table-lookup interpolation to reduce computation and execution time. When designing table-lookup interpolation, application writers face various choices such as table size and interpolation mechanism (linear, quadratic, etc). The better the interpolation, the more intensive the computation, and as a result, often computationally least expensive mechanism is chosen, compromising quality for performance.

In our project, we characterize the performance a particular kind of streaming application: sound synthesis oscillator. We determine the existing performance bottlenecks in this application, and investigate the effectiveness of using Streaming SIMD Extension (SSE) instruction set in this application. We demonstrate that the SSE-enhanced code can deliver performance improvement when ratio of computation to memory operations is favorable. In particular, in case of quadratic interpolation, we show a 27% performance increase as compared to the best alternative version that does not use SSE. Further, we demonstrate that the additional penalty of using quadratic interpolation is reduced to 11% as compared to linear interpolation, when using SSE. This compares favorably to 50% penalty in case when SSE is not used. This provides a much more favorable trade-off in the application design space.

Introduction

Table-lookup interpolation is an old but still important technique in many computation-intensive programs like sound synthesis [Roads], key-frame animation [Burtnyk+], and scientific computations [Ueberhuber]. The technique is especially effective in replacing the computation of expressions where there is limited range of input values and such expressions are costly to evaluate directly. The most common usage of table-lookup interpolation is to restore the curve of the continuous signal from limited amount of sampled data. However, there are a lot of factors can affect the performance of this straightforward method.

As normally it is impossible to restore the exact signal, table-lookup interpolation introduces the error which results in noise. Error can be reduced by increasing the table size and/or by increasing the quality of the interpolation. But both of these also affect the signal computation cost. In addition, general purpose processors like Pentium series are not particularly good at either random access to main memory or interpolation, but these operations are at the core of table-lookup interpolation.

So a detailed study and structure design are required to carefully choose the trade-offs between those key factors while still achieving satisfactory quality output.

In this paper we are focusing on the table-lookup interpolation for sound synthesis. In sound synthesis applications, table-lookup interpolation is used to generate sinusoids for additive synthesis, and fixed spectra for group additive, spectral interpolation, and vector synthesis. Samplers also use a form of table-lookup interpolation, and sample-rate conversion can be viewed as a generalization of table-lookup interpolation. Thus, a fundamental operation for a variety of synthesis techniques is reading samples from tables and doing interpolation between samples.

A case in point is table-lookup oscillator. This case is particularly interesting because it is one of the most fundamental techniques in sound synthesis applications, and it also well represents a typical process of generating sound samples [Moore].

We evaluated the performance of the table-lookup oscillator, analyzed its characteristics and explored possible improvements to the performance by using standard optimization techniques. Furthermore, we investigated the possibility of utilizing the multimedia extensions of modern architectures, the Streaming SIMD Extension (SSE) in Pentium III, to further improve the performance of such sound synthesis applications [SSE], as past research [Barbic+] has demonstrated the performance benefits of using MMX / SSE to streaming multimedia applications.

The Problem

The purpose of a table lookup oscillator is to output samples of a periodic function. The typical loop for computing samples, given frequency *hz*, sample rate *sr*, and lookup table *table* is:

```
while ( stream )
{
       i = (int) phase;    // get the integral part of the phase
       f = p - i;          // get the decimal part of the phase
       // do lookups
       s1 = table[i];
       s2 = table[i+1];
       output(s1 * (1-f) + s2 * f);  // linearly[1] interpolate the value
       phase += increment; // increment = hz/sr
       // phase is within the range of the lookup table
       phase = phase MOD Sizeof(Table);
}
```

In our experiment, the lookup table is generated by evenly sampling 2048 data points from a complete period of a sinusoid. A previous study, [Dannenberg], shows that 2048 is a suitable table size combining with linear interpolation to produce the sound with satisfying quality. As the frequency and sample rate doesn't affect the actual computation effort in the loop, we pre-define the sample rate *sr* as 44.1 KHz and frequency *hz* as 261.62Hz (the note Middle C) before entering the loop. In order to make sure the program executing time is long enough to be sampled and analyzed, we also define the duration of the generated sound to be 1000 seconds, so that the key loop will run approximately 44 million times every time the program gets executed.

[1] This version performs linear interpolation. Quadratic interpolation code is similar, but uses three lookup values and a quadratic formula to compute the sample.

Performance Analysis

We started out with the code extracted directly from the sound synthesis language *Nyquist* [Dannenberg] (the **original** version, as we refer to it throughout the paper). First we need to determine whether the bottleneck is memory bound or CPU bound. We analyzed the program through *VTune Perforamnce Analyzer* [VTune] and found out that the Level 1 cache misses is less than 1% during the running time. It is relatively easy to understand since the lookup table contains only 2048 floating point values, so normally the whole table can be fit into the cache. However, the preliminary sampling and profiling results shows that it has low level of the instruction level parallelism. In *VTune*, the cycle per instruction (CPI) of the program is only around 2.7 (note that this data alone can not be that meaningful – however, a simple test program doing similar floating point computation, but without critical path dependencies, achieved lower CPI) . So, while memory is not a bottleneck, CPU is not fully utilized.

There are several factors that may cause the CPU to be underutilized. First, the interpolation is applied on floating point values, and floating point operations are computationally expensive.

Secondly, one of the key operations, the floating point to integer conversion (FTOL) routine, turns out to be the most expensive computation instruction in the loop [Herf], [de Castro Lopo]. In our base version compiled with *Visual C++, VTune* analyzer results suggest that 50 CPU[2] cycles are needed for one single FTOL operation, which is outrageous, considering the wide usage of the FTOL operation in many applications such as audio, video and graphics processing.

It is mainly because there is no dedicated x86 instructions to do the floating point to integer conversion. There is an instruction on the chip that does a conversion (*fistpl*), but it respects the chip's current rounding mode. The default rounding mode is to round the floating point value to the closest integer, as it is required for all normal operations like addition, subtraction, multiplication etc. So in order to implement the "correct" version of truncation for the float to int cast, it needs to switch the rounding mode to the truncation mode before call to *fistpl* and once the conversion is finished, the mode needs to be switched back.

A typical assembly interpretation of the FTOL operation is shown as below.

```
fnstcw -2(%ebp)         ; store FPU control word
movw -2(%ebp),%di       ; move FPU control word to di register
orw $3072,%di           ; modify di
movw %di,-4(%ebp)       ; move di to the stack
fldcw -4(%ebp)          ; load same value from stack into FPU control word
fistpl -8(%ebp)         ; store floating point value as an integer on the stack
movl -8(%ebp),%eax      ; move the integer value from stack to eax
fldcw -2(%ebp)          ; restore FPU control word
```

The instruction which causes the real damage is *fldcw* (FPU load control word). Whenever the FPU encounters this instruction it flushes its pipeline and loads the control word before continuing operation. The FPUs of modern CPUs like the Pentium III, Pentium IV and AMD Athlon rely on deep pipelines to achieve higher peak performance. So this piece of code reduces

[2] Including stall time resulting from flushing the floating point state can potentially bring the cycle count due to FTOL to as much as 70 cycles. A previous study [Herf] found FTOL to be responsible for as much as 80 cycles; however, CPU model was not mentioned. At any rate, FTOL is expensive, and this is confirmed independently [Herf], [de Castro Lopo].

the floating point performance of the CPU to level of a non-pipelined FPU. Also another important instruction *fistpl* that does the actual conversion work also requires about 6 CPU cycles to complete.

There is also significant performance difference between the *Visual C++* compiled version and the *GCC* compiled version. We found out that *Visual C++* defines the FTOL routine in standard library and the function call is not in-lined, which results in significant function call overhead, while in *GCC*, the FTOL function is in-lined. But overall the floating point to integer conversion is still quite a bit expensive.

The third factor that should be noticed is the critical path in the loop. Considering the dependencies between each operation, the floating point multiplication in the interpolation must wait for the outcome of the corresponding table lookups, and table lookups must wait for the floating point to integer conversion. This chain of dependencies may leave the CPU underutilized.

Optimizations

Having characterized the performance of the table-lookup oscillator code and identified the bottlenecks, we proceeded to optimize the code. Since the floating point to integer conversion took significant amount of execution time, we first attempted to find a more optimized alternative to the compiler provided conversion routine.

Optimizing Floating Point to Integer Conversion

Survey of the floating point conversion topic revealed that there are several alternatives. We will describe each of these alternatives separately.

Alternative One: Not Using Floating Point Conversion (NO_FTOL)

Analysis of the critical loop shows that it is possible to re-write the code without using floating point to integer conversion. Indeed, the conversion is needed for variable `phase`. `phase` is incremented by a constant amount `phase_inc` every iteration of the loop. Thus, by separating both `phase` and `phase_inc` into integer and decimal-only parts, and performing the necessary arithmetic individually on each part, we can easily avoid floating point to integer conversion. All we need to do is ensure that decimal part overflow is handled, and since overflow can only be 0 or 1, the overhead of overflow handling is only one branch operation. While this solution is very efficient, it is not as general. For example, if any part of the computation of variable `phase` required floating point multiplication or several additions in a sequence, handling overflow would not be so easy.

Alternative Two: Using Fixed Point Arithmetic (FIXP)

An alternative to floating point arithmetic is fixed point arithmetic. The idea of fixed point arithmetic is to allocate several bits to store the integer part, and the rest of the bits to store the decimal part. Since the integer part of variable `phase` is mod-ed by the size of the table, we can easily determine how many bits to allocate to the integer part. In our experiments the table size is equal to 2^11 = 2048, thus we allocated 11 bits to the integer part, and 53 bits to the decimal part. Notice that 53 bits ensure very high precision for the decimal part.

Recall computation of next value of variable `phase` requires only addition. Thus, fixed point arithmetic works well. If computation of `phase` contained multiplications, this alternative would not work.

While fixed point arithmetic performance better than the original version, it does not outperform other, better alternatives.

Alternative Three: Using Intel Rounding Instruction Directly (LRINT)

Performance analysis of the original version demonstrated, that large part of the overhead of the compiler-provided floating point conversion was due to setting and resetting the rounding mode of the FPU, before `fistpl` instruction is executed. Assuming that the FPU is set to round, as opposed to truncate, this method will work: `Floor[x] = Round[x-0.5],`

This alternative still utilizes the `fistpl` instruction, and therefore is somewhat expensive.

Alternative Four: Manipulating FP Bit Structure (REAL2INT)

There are several methods in this category. All of these methods use the IEEE floating point representation format and manipulate the exponent and the mantissa bits accordingly to truncate the floating point number. We know of at least three methods. According to [Herf], Real2Int works best in terms of performance and precisely complies to the ANSI standard for floating point conversion, provided the FPU is in double-precision mode, which is a reasonable assumption. The listing of the Real2Int code is in the appendix.

Since this alternative is general enough, and is very fast, we chose it as a starting point for further optimizations. We call this the **benchmark** version.

Further Optimizations: Loop Unrolling and Software Pipelining

Analysis of the music synthesis code shows that there are dependencies in the computation, possibly causing performance bottlenecks. The loop computation has roughly the following structure.

```
A.     phase = update (phase)
B.     index = convert(phase)
C.     s1 = load table[index]
D.     s2 = load table[index+1]
E.     result = compute(phase, s1, s2)
F.     store result
```

The critical path is as follows:

```
A -> B
B -> C, B -> D
C, D -> E
E -> F
```

Getting rid of the critical path can potentially improve performance, by keeping the pipeline fully utilized. We tried two methods: software pipelining, and loop unrolling.

Software Pipelining (REAL2INT_SOFT_PIPE)

The idea of software pipelining [Patterson+] is to allow for more instruction-level parallelism by executing in the same iteration of the loop operations that are not dependent. To do so, different iterations of each operation are executed. Thus, the code in the loop will look like this:

```
A_5
B_4
C_3, D_3
E_2
F_1
Perform several copies
```

In this notation, A_5 for example, corresponds to computing the value of phase for the 5^{th} iteration of the loop. Notice that in order to allow for software pipelining, we need to keep copies of the same variable from several iterations of the loop, and perform copies.

This approach potentially increases performance, because the dependencies are removed. However, the additional copies will add to the execution time, decreasing performance gains.

Software pipelining on top of Real2Int version did achieve performance improvements of about 8% (however, this was observed for only *GCC* compiler).

Loop Unrolling (REAL2INT_UNROLL_4)

Another way to increase the parallelism of the program is to unroll the critical loop. We did manual loop unrolling twice and four times. Though loop unrolling has potential performance advantages, it can increase register pressure, especially considering the fact that the loop already has significant number of floating point variables (the pseudo-code of the loop does not show all the variables).

Unrolling the loop 4 times on top of Real2Int version improves performance by about 15%, both for *Visual* C++ and *GCC*. Notice that the performance of unrolling the loop 4 times was slightly better than that of the software-pipelined version.

Optimizations Using SSE

Intel Pentium III processor provides performance support for streaming multimedia applications by means of single instruction, multiple data (SIMD) operations. This extension, called SSE (Streaming SIMD Extension), provides 8 128-bit registers and instructions that allow arithmetic and logical instructions on 4 32-bit values.

Modern compilers do not provide optimizations that take advantage of SSE. Thus, in order to take advantage of SSE, we needed to hand-write code.

Since SSE instructions operate on 4 32-bit packed floating values, it is natural to choose as a starting point the unrolled 4-times version of the code.

We tried two approaches for writing the SSE code. First, we tried writing inline assembly. Unfortunately, among the instructions we use, there is one that is privileged, and caused some problems while compiling with *Visual* C++. Instead, we tried a second approach, which took advantage of a special fix-pack to *Visual* C++. This fix-pack provided support for writing SSE

code by means of C-language primitives that get directly translated into SSE assembly. We show one sample of SSE-enhanced code in the appendix.

Performance Analysis of the SSE version

The first version of the SSE-enhanced table-lookup oscillator code ran approximately as fast as the benchmark version, and thus significantly slower than the unrolled 4-times version. The primary bottleneck is the set of memory operations required to pack the SSE registers. In order to load the SSE registers, the 4 floating point values need to be aligned on a 16-byte (128 bit) boundary continuously. The data for different iterations of the loop that is involved in the interpolation computation is not laid out contiguously in memory, thus we need to perform 4 loads, 4 stores and then an SSE register load to prepare each packed value. We have a total of 4 variables involved. Thus a total of 32 load / stores and 4 SSE register loads are required. These memory operations start to dominate, especially considering that the linear interpolation case does not require as much computation. Unfortunately, with the exception of variable `phase`, it is not possible to avoid loads and stores for any variable. In particular, for S1 and S2, which are the endpoints of interpolation, the four different values come from non-consecutive locations in the lookup table. Thus, it is not possible to pack four consecutive values of S1 (and same for S2) together, without doing an additional load and store for each. The same is true for the integer value of `phase`, which is also used in computation.

We manually re-ordered instructions with the hope of interleaving some computation with memory operations. However, the improvements were small (just under 4%).

Quadratic version of the interpolation benefits more from the SSE support. This is expected, as the computation to memory operation ratio is significantly higher.

Experiments and Results

We performed several experiments, with code running linear and quadratic interpolation. Quadratic interpolation version is interesting for two reasons. First, quadratic interpolation can generate higher quality sound. However, it is not preferred because in normal case, it is much slower than linear interpolation. Second, because quadratic interpolation has significantly more computation, it can potentially benefit more from SSE-enhancement. Below we report the results of these experiments.

Experiments were performed on a CMU CS-facilitized Pentium III, running at 1 GHz. A number of the experiments were performed using code compiled by both *Visual* C++ and *GCC* compilers. Using two different compilers ensures that the performance is not significantly influenced by the choice of the compiler. With the exception of the very original version, which used the compiler-provided floating point to integer conversion, we saw very similar results for both *Visual* C++ and *GCC* compilers. Since we are interested in the execution time of the code, we report results produced by *Visual* C++ profiler for *Visual* C++ compiled version of the code and cygwin-provided time command for *GCC* -compiled version. However, we also took advantage of Intel's *VTune* performance profiler to do the bottleneck analysis of various versions. In particular, *VTune* enabled us to measure L1 cache misses, branch miss-predicts, cycles per instruction for the entire code and for critical routines such as __FTOL (floating point to integer conversion), cycles per __FTOL, µ-ops executed per cycle.

Experiments for all versions ran the code to generate a sinusoid sound in Middle C for 1000 seconds at 44,100 Hertz. Thus, the critical loop was executed 44,100,000 times in each case. Since the speed of the CPU is 1GHz, the running time can be used to compute number of cycles taken to execute the code. Notice that this information is also available from *VTune*.

To give the reader an idea of why optimization of the loop may be difficult, consider the number of cycles taken by one iteration of the loop of the Real2Int version, which is the benchmark for all our SSE optimizations. For linear interpolation, Real2Int took approx. 1,300,000,000 cycles, which divided by approx. 44,100,000 iterations, gives approx. 29 cycles per loop iteration.

Figure 1 presents the results for linear interpolation. First, note the difference in *Visual* C++ and *GCC*-compiled performance of the Original version. The additional time for the *Visual* C++ version is due to the fact that the floating point to integer conversion is implemented as a non-inline function. Further, notice that codes using the optimized floating point to integer conversions do much better in all cases. Notice that No_Ftol version performs best. Notice that the SSE_u4_v2, runs in 1250 ms, slightly better than Real2Int, which runs in 1300 ms (for *Visual* C++). However, Real2Int_unroll_4 still runs better with 1100 ms running time for *Visual* C++.

Figure 2 shows experiment results of the quadratic interpolation code. The SSE-enhanced version achieves 27% speedup over the Real2Int version.

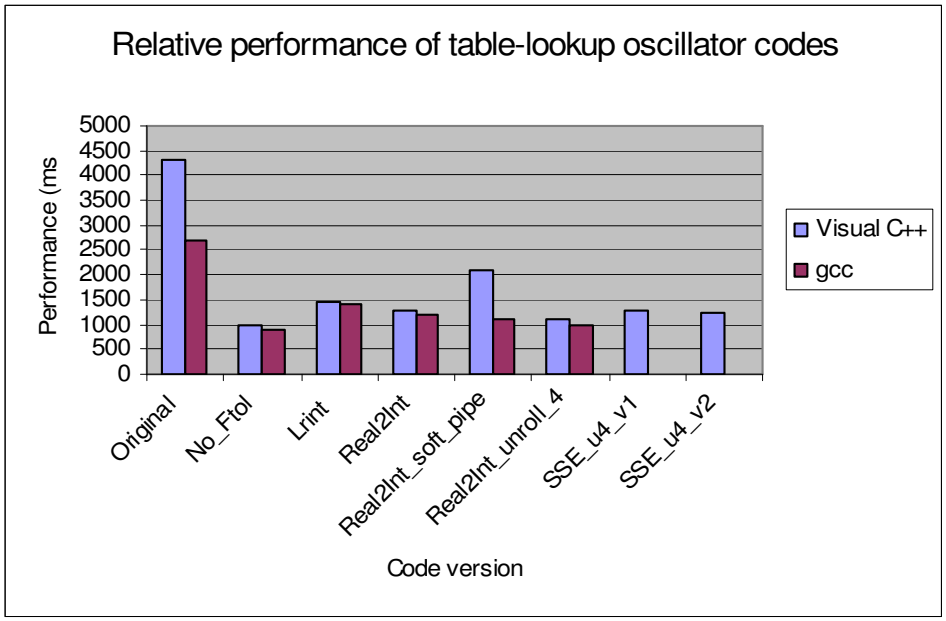

Figure 1. *Performance of table-lookup oscillator codes, using linear interpolation. Tests are run on Pentium III, 1 GHz machine. Measurements represent execution time. Results reported show the average of 3 trials out of 4 trials. Results of the first trial discarded. Variance was insignificant. SSE code available for VC++ only.*

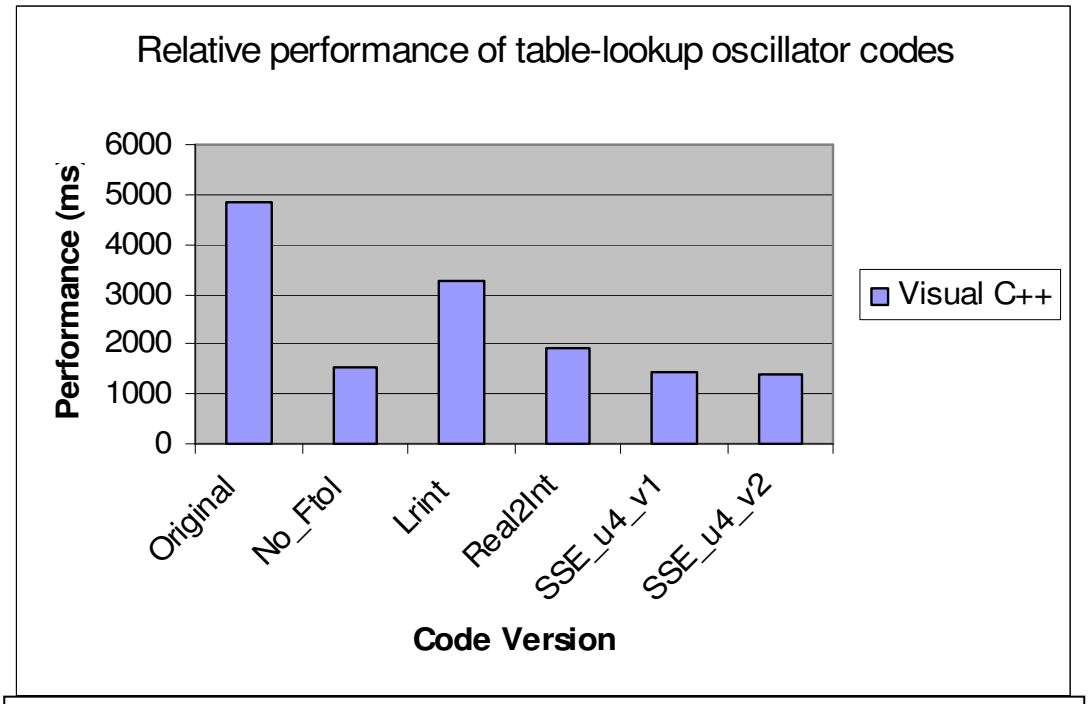

Figure 2. *Performance of table-lookup oscillato codes, using quadratic interpolation. Tests are run on Pentium III, 1 GHz machine. Measurements show execution time. Results show the average of 3 trials out of 4 trials. Results of the first trial discarded. Variance observed was insignificant.*

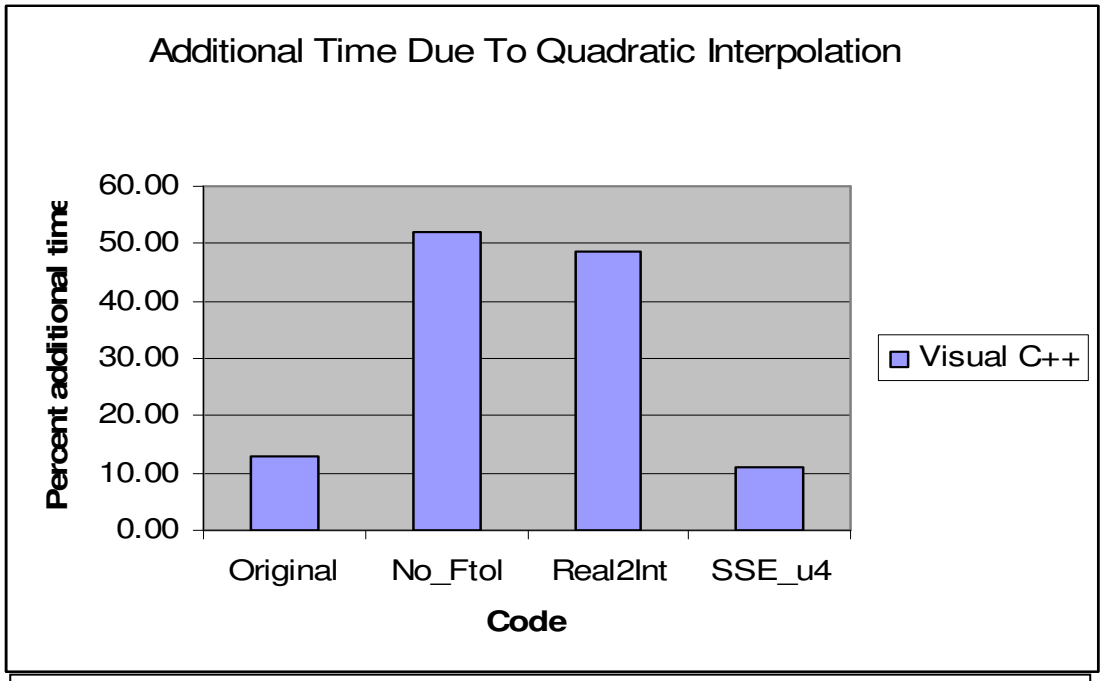

Figure 3. *Additional time, expressed as a percentage, to run the same version of the code using quadratic interpolation (as opposed to linear). Notice that in case of No_Ftol and Real2Int, the penalty of using quadratic interpolation is very high, about 50 percent. However, SSE-enhanced code adds very little overhead for quadratic.*

Figure 3 presents an interesting aspect of the experiment results. As mentioned above, quadratic may not be preferred, because the performance penalty compared to linear interpolation is somewhat high. We show the additional percentage of time due to quadratic interpolation for 4 cases: (1) original version of the code, (2) code with no floating point conversion (No_Ftol), (3) code using Real2Int (Real2Int), and (4) code with best SSE version for quadratic and linear (SSE). Notice that in cases 2 and 3, performance penalty due to quadratic interpolation is about 50 percent. Using SSE, we are able to bring that down to 11 percent. Using SSE encourages use of quadratic interpolation, which is expected.

Contribution of the Work and Conclusions

At the start of the project, we set a goal of characterizing the performance of a typical piece of sound synthesis code and attempting to improve the performance using conventional optimizations and SSE. Our second goal has been to evaluate the performance benefits of SSE to sound synthesis applications, which typically use combination of lookup tables and interpolation to process stream of data.

For our first goal, we were able to achieve performance improvement of 70-75%, or a factor of 3.4 – 4 reduction in speed, as compared to the original version of the code. However, a large portion of that performance gain is attributed to implementing a more optimized version of floating point to integer conversion routine. In the case of linear optimization, SSE-enhanced code did not deliver performance improvement compared to the benchmark version chosen (Real2Int). In fact, SSE-enhanced version was slower than the best version we could produce using optimizations other than SSE (loop unrolling, software pipelining). In case of quadratic interpolation, which exhibits a higher level of computation to memory operation ratio, we were able to achieve 27% performance improvement by using SSE, as compared to the benchmark version.

Our results show that using single-instruction, multiple-data instructions can achieve performance improvement, if ratio of computation to memory operations is significant. In particular, for certain streaming-data applications, the extent to which the code is wells-suited for packing the data can determine the extent of performance benefits of SSE. In particular, applications which take advantage of table lookups may not benefit from SSE as much, since preparing and packing values from different (non-contiguous) entries of the table take up significant time.

Another interesting result is that the performance of the quadratic interpolation version relative to the linear version decreases, when using SSE. This encourages use of quadratic interpolation which is desirable.

From engineering point of view, hand-crafting SSE code can be tedious and error prone. Because compilers do not provide support for SSE code generation, SSE-enhanced code is not very portable across compilers. It will be a valuable contribution to provide automatic translation of conventional arithmetic expressions into SSE code.

Acknowledgements

We would like to thank Professor Seth Goldstein and Doctor Roger Dannenberg for help in this project. Roger provided an interesting topic for the project, granted us access to music synthesis source code and shared with us his experience with analyzing the performance of the code. Seth helped us formulate a goal, scope the work, and provided critical guidance during the project. Also, we would like to thank our colleagues Jernej Barbic, Bianca Schroeder, and Minglong Shao, who helped us with choosing and getting familiar with performance analysis tools and tips in interpreting performance results.

References

[Barbic+] Barbic, Rosenkrantz, Potetz, "Evaluating the Impact of Pentium III Multimedia Enhancements on JPEG Decoding." CMU CS740 Computer Architecture Project, Fall 2001.

[Burtnyk+] N. Burtnyk and M. Wein. "Computer generated keyframe animation," *Journal of the Society of Motion Picture and Television Engineers*, 80(3):149-53, March 1971.

[de Castro Lopo] Erik de Castro Lopo. 2001. "Faster Floating Point to Integer Conversions", electronic version of the paper available online at *http://mega-nerd.com/FPcast/*

[Dannenberg 1] Dannenberg, "Interpolation Error in Waveform Table Lookup," in *Proceedings of the 1998 International Computer Music Conference,* (1998), pp 240-243.

[Dannenberg 2] Dannenberg, *Nyquist, a Sound Synthesis and Composition Language,* software and information available online at *http://www-2.cs.cmu.edu/~music/nyquist/*

[Patterson+] Patterson; Hennesy. 2002. *Computer Architecture: A Quantitative Approach.* San Francisco, CA: Morgan Kauffman.

[Herf] Herf. 2000. "Know your FPU", *electronic version of the paper available online at http://www.stereopsis.com/FPU.html*.

[Intel] Intel Architecture Software Developer's Manual: Volume 2, Instruction Set Reference. Intel Corporation, 1999.

[Moore] Moore, F. R. 1977. "Table lookup noise for sinusoidal digital oscillators," *Computer Music Journal* 1(2), pp26-29. Reprinted in C. Roads and J. Strawn, eds. 1985. *Foundations of Computer Music.* MIT Press. pp. 326-334.

[Roads] Roads, C. 1996. *The Computer Music Tutorial.* Cambridge: MIT Press, p. 93.

[SSE] Introduction to Streaming SIMD Extension: a Tutorial, Intel Corporation, 1999. *http://www.intel.com/software/products/college/ia32/strmsimd/clikngo.htm*

[Ueberhuber] C.W. Ueberhuber, *Numerical Computation (Volume 1), Chapter 9 "Interpolation"*, Springer (1997), ISBN 3-540-62058-3

[VTune] Intel Corporation, *VTune Performance Analyzer*, software information available at *http://www.intel.com/software/products/vtune/*

Appendix: Code Snippets

Real2Int: custom routine for floating point to integer conversion

```
//  ***************  custom float to integer conversion routine

typedef double lreal;
typedef float  real;
typedef unsigned long uint32;
typedef long int32;

const lreal _double2fixmagic = 68719476736.0*1.5;     //2^36 * 1.5,  (52-
_shiftamt=36) uses limited precisicion to floor
const int32 _shiftamt        = 16;                    //16.16 fixed point
representation,

#if BigEndian_
       #define iexp_                     0
       #define iman_                     1
#else
       #define iexp_                     1
       #define iman_                     0
#endif //BigEndian_

// ====================================================================
// Real2Int
// ====================================================================
inline int32 Real2Int(lreal val)
{
       val          = val + _double2fixmagic;
       return ((int32*)&val)[iman_] >> _shiftamt;
#endif
}
```

osc_fetch_sse: SSE enhanced version of osc code

```
sample_type *osc__fetch_sse_unroll_four(register osc_susp_type susp)
{
    int cnt = 0; // how many samples computed
    int togo;
    int n;
    register sample_type *out_ptr;
    register sample_type *out_ptr_reg;

    register double ph_incr_reg;
    register sample_type * table_ptr_reg;
    register double table_len_reg;
    register double phase_reg;
    register double phase_reg_one;
    register double phase_reg_two;
    register double phase_reg_three;

    out_ptr = (sample_type*)malloc(sizeof(sample_type)*(susp->terminate_cnt+1));
    // first compute how many samples to generate in inner loop:
    togo = susp->terminate_cnt - cnt;

    n = togo;
    ph_incr_reg = susp->ph_incr;
    table_ptr_reg = susp->table_ptr;
    table_len_reg = susp->table_len;
    phase_reg = susp->phase;
    out_ptr_reg = out_ptr;

       __m128 x1;    //  these are allocated to sse registers
       __m128 x2;
       __m128 phase;
       __m128 index;
       __m128 result;
```

```
      register floatPacked X1;
      floatPacked Phase;
      register floatPacked Index;

   if (n) do{             // the inner sample computation loop
            long table_index;
            long table_index_one;
            long table_index_two;
            long table_index_three;

            table_index = Real2Int(phase_reg);
            Index.f0 = table_index;           //  convert zero
            phase_reg_one = phase_reg + ph_incr_reg;       // increment one
            phase_reg_two = phase_reg_one + ph_incr_reg;   //  increment two
            phase_reg_three = phase_reg_two + ph_incr_reg; //  increment three

            Phase.f0 = phase_reg;

            if ( phase_reg_three >= table_len_reg)
            {
                  phase_reg_three -= table_len_reg;
                  if ( phase_reg_two >= table_len_reg)
                  {
                        phase_reg_two -= table_len_reg;
                        if ( phase_reg_one >= table_len_reg)
                              phase_reg_one -= table_len_reg;
                  }
            }

            table_index_one = Real2Int (phase_reg_one);    // convert one
            table_index_two = Real2Int (phase_reg_two);    //  convert two
            table_index_three = Real2Int (phase_reg_three);//  convert three
            Phase.f1 = phase_reg_one;
            Phase.f2 = phase_reg_two;
            Phase.f3 = phase_reg_three;
            phase = _mm_load_ps((float*) &Phase);

            Index.f1 = table_index_one;
            Index.f2 = table_index_two;
            Index.f3 = table_index_three;

            X1.f0 = table_ptr_reg[table_index];            //  **  load zero
            X1.f1 = table_ptr_reg[table_index_one]; //  **  load one
            X1.f2 = table_ptr_reg[table_index_two]; //  **  load one
            X1.f3 = table_ptr_reg[table_index_three];      //  **  load one
            index = _mm_load_ps((float*) &Index);
            x1 = _mm_load_ps((float*) &X1);

            X1.f0 = table_ptr_reg[table_index+1];          //  **  load zero
            X1.f1 = table_ptr_reg[table_index_one+1];      //  **  load one
            X1.f2 = table_ptr_reg[table_index_two+1];      //  **  load one
            X1.f3 = table_ptr_reg[table_index_three+1];    //  **  load one

            x2 = _mm_load_ps((float*) &X1);
            x2  = _mm_sub_ps( x2, x1);
            index = _mm_sub_ps(phase, index);
            index = _mm_mul_ps(index, x2);
            result = _mm_add_ps(x1, index);
            _mm_store_ps((float *) out_ptr_reg, result );
            out_ptr_reg +=4;
            phase_reg = phase_reg_three + ph_incr_reg;

            while (phase_reg >= table_len_reg) phase_reg -= table_len_reg;
            n -= 4;
   } while (n>0); // inner loop

   susp->phase = phase_reg;
   cnt += togo;
   susp->susp.current += cnt;

      return out_ptr;
} // osc__fetch_sse_unroll_f
```