Real-Time Software Synthesis on Superscalar Architectures[1]

Roger B. Dannenberg and Nick Thompson
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
Email: dannenberg@cs.cmu.edu, nix@cs.cmu.edu

Advances in processor technology are making it possible to use general-purpose personal computers as real-time signal processors. This enables highly-integrated “all-software” systems for real-time music processing. Much has been speculated about the behavior of software synthesizers, but there has been relatively little actual experimentation and measurement to verify or refute the “folklore” that has appeared. In the hopes of better understanding this important future technology, we have performed extensive measurements on several types of processors. We report our findings here and discuss the implications for software synthesis systems.

Introduction

Superscalar architectures are expected to compute 500 to 1000 million instructions per second (MIPS) by the end of the decade. Software synthesis on superscalars will offer greater speed, flexibility, simplicity, and integration than today's systems based on DSP chips. In this paper, we describe the advantages of the superscalar architecture and indicate its future potential. We outline the requirements that this architecture places on software implementations and how these requirements are met.

Superscalar Architecture

Superscalar processors represent the state of the art in computer architecture. Current examples include the Intel Pentium and IBM/Motorola PowerPC processors. These machines feature single-cycle execution of common instructions, the issue of multiple instructions per cycle, and multiple pipelined arithmetic units. Instruction scheduling in the compiler assures that many floating point operations are computed in parallel. By the year 2000, we expect personal computers will deliver performance we now associate with super-computers. This means that real-time signal processing applications may no longer require special-purpose hardware or digital signal processors. These applications can be supported in a single, integrated, high-performance programming environment.

[1] Published as: Roger B. Dannenberg and Nick Thompson, “Real-Time Software Synthesis on Superscalar Architectures,” *Computer Music Journal*, 21(3) (Fall 1997), pp. 83-94.

There is, however, some debate over the viability of superscalars for signal processing. First, these systems rely on a memory hierarchy with caching at various levels to provide instructions and data to the CPU. This is good in that it provides the programmer with a very large flat address space, but caching makes performance hard to predict relative to DSPs. Second, an integrated system requires real-time support from the operating system, yet most operating systems provide weak support (if any) for real-time applications. Third, cost will be an important factor until personal computers are faster than low-cost plug-in DSP systems.

Nevertheless, we believe that it is only a matter of time before DSPs for computer music are obsolete. The i860-based IRCAM IMW (Lindemann, *et al.* 1991) is a major milestone in this progression, but even the IMW has the flavor of an add-on DSP system. It has multiple processors, a specialized operating system, and is hosted by a non-real-time NeXT computer. (However, much of this environment recently has been ported to a uniprocessor.) Vercoe's Csound (Vercoe and Ellis 1990) running on a DEC workstation is a better illustration of the "all software" approach we believe will soon be the norm. Csound illustrates the flexibility and portability of the software approach along with impressive performance. Adrian Freed (1994) is also exploring the possibilities of software synthesis in his HTM environment. Freed (1993) addresses many issues of optimizing signal processing software. Our work in this area began in 1983 with the design of Arctic (Dannenberg, McAvinney, and Rubine 1986), a very high-level language for real-time control. Arctic showed how a single language could integrate note-level event processing, control-signal generation, and audio synthesis.

The language Nyquist (Dannenberg, to appear) is based on Arctic. Nyquist offers a high-level and general treatment of scores, synthesis algorithms, and temporal behavior. Superscalar processors seem ideal to handle the mixture of symbolic and signal processing required by Nyquist. Although Nyquist has developed into a non-real-time system, it has provided a good platform for experimentation, and high performance signal processing is just as important in non-real-time settings. In the design of Nyquist, we set out to answer the following questions: What characteristics of superscalars are important for music synthesis? What new techniques are necessary to maximize performance? What is the overhead or benefit of the advanced features of Nyquist?

Agarwal *et al.* (1994) discuss various optimizations for numerical processing, and our work is strongly influenced by these ideas. Our work focusses specifically on software for audio signal processing.

Where does CPU time go?

To optimize any software system, the first step is to find out where computation time is being spent and why. With pipelined hardware, multiple levels of memory hierarchy, and optimizing compilers, this can be a difficult job. To start, let us consider where time might be spent in software synthesis:

Numerical Processing.

At the core of nearly every signal processing algorithm are multiply and add operations. These take time in two senses. First, there are a limited number of arithmetic units available on every processor cycle, so arithmetic operations must wait for an available unit. Second, most arithmetic units take several machine cycles to produce results, so even after the operands are delivered to a unit, the machine may stall with nothing to do until the result appears. Numerical processing time gives a sense of the lower bound on execution time.

Loads and Stores.

While operations are typically performed on data in registers, operands are often stored in primary memory because audio signals have many more samples than would fit into registers. Consequently, many instructions are required to move data incrementally into registers and back out to memory. To increase efficiency, one or more levels of fast cache memory hold copies of recently accessed memory locations. To the extent that cache memory holds the data required, load and store operations run much faster. Techniques that optimize the cache will lead to faster computation. This is often the source of software architecture decisions.

Instruction Parallelism.

Superscalar processors fetch multiple instructions on every cycle. There are multiple arithmetic, logic, and branch units available to process these instructions. If the instruction mix is right, many instructions can be executed simultaneously. If the instruction mix is wrong, or if each instruction depends upon results from the previous one, the instructions will be executed sequentially and many cycles will be “wasted” waiting for results. Instructional parallelism can have dramatic implications for software architecture. For example, if loop overhead is zero (loop instructions run in parallel with numerical computation), then small loop bodies can be as efficient as large ones.

Algorithmic Optimization.

Every instruction set has its tricks. In general, we would like to avoid too much reliance on processor-specific optimizations because we are interested in high-level portable languages. If we really wanted to code at the instruction level and ignore portability, it would make more sense to take a DSP-based approach. Nevertheless, a look at compiler-generated assembly code can give us some idea of how far portable code is from optimal code. Also, there are some algorithmic optimizations that appear to be quite portable and worth using.

Optimizing Software Architecture

The places where processors spend time are well-understood, but this does not mean it is easy to optimize programs. In many cases there are tradeoffs, and it is not obvious which factor is most important. In this section, we will consider some design decisions that lead to a variety of software organizations. We will develop a set of specific questions. In the following sections we will describe experiments motivated by these questions and present experimental results.

One design decision concerns modularity. Typically, sound synthesis systems offer many processing elements or unit generators that perform simple operations such as function generation, filtering, addition, and multiplication. These operations are combined to create “instruments” or complex signal processors. To combine these operation, they are typically linked by memory buffers we will call *sample blocks*.

How big should sample blocks be? On one hand, blocks should be small so that they all fit in the cache, insuring fast access. On the other hand, there is overhead associated with invoking a signal operation, including procedure calls, loading sample addresses into registers, and loading constants required for the operation. These costs, which occur once per operation invocation, can be amortized across many samples if large blocks are used.

Another decision is: how much work should be performed in each inner loop? If two operations can be merged into one, there is less loop overhead, a sample block is eliminated, and the stores and loads required to save data to the block and then retrieve it are eliminated. On the other hand, the larger the inner loop, the more registers are required. Registers typically store pointers to data, filter coefficients, phase increments, and other values. If the inner loop is too large, some of these values will not fit in the available registers, and additional loads and stores will be required.

Assuming that sample blocks are good, what should be done with low-sample-rate control signals? How much could be saved by computing control-rate signals in blocks? Would linear interpolation allow lower control rates and less computation, or would the overhead of interpolation generate a net increase in computation time?

It turns out that inner loop computations dominate the total computation time. What can be done to compute samples more efficiently? A few techniques are explored to gain some insights.

Table lookup is a problem for any memory system because random accesses to large tables will often cause a cache miss. How big is this problem?

To answer these questions, we implemented and measured a number of synthesis strategies. Our initial work produced answers, but many of them were counter-intuitive. Analyzing compiler output gave us some clues, but said nothing about cache behavior and instruction parallelism. Ultimately, we resorted to instruction-level timing measurements to determine exactly where time was spent. Our results are surprisingly consistent across widely varying architectures, including the PowerPC and Pentium processors. However, we can only speculate (and hope) that the trends we observe will continue for the next 5 to 10 years and beyond.

Block Size

We used Csound to perform an additive synthesis task and varied the block size. Block computations give dramatic speedup: Csound improves by a factor of about 7 with large blocks (see Figure 1.). We did not rewrite our code for the special case of blocksize = 1, so in reality the speedup would not be so great. Nevertheless, the trend is clear that blocks do amortize the cost of operation invocation and loop setup over many samples.

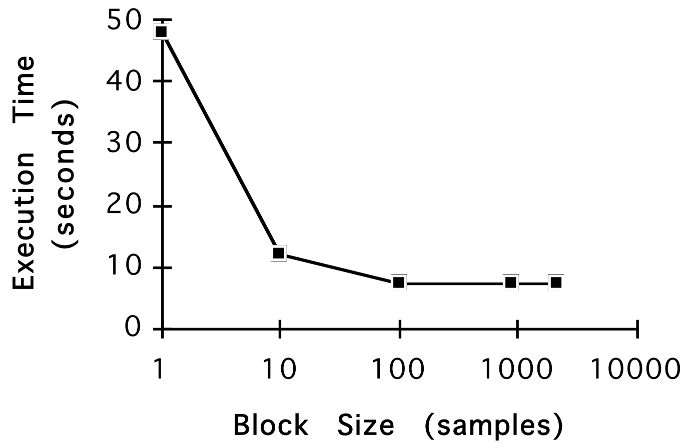

Figure 1. Performance of Csound as a function of block size. Very large blocks do not degrade performance, even though the large blocks do not fit in the cache.

Is caching critical to performance? If so, we would expect small block sizes (relative to the cache) to give higher performance. In that case, the curve in Figure 1 would be U-shaped, rising on the left due to loop overhead, and rising on the right due to cache effects. Although the curve does rise slightly with very large block sizes, the change is small. This indicates that any data cache effect is minuscule (Dannenberg and Mercer 1992).

How can this be? After all, a cache miss costs many cycles. Surely a purposeful attempt at achieving poor cache performance should show some effect. To understand the good performance we observed, it is necessary to consider what happens when a load instruction requests a memory word that is not in the cache. First, the word is loaded from primary memory (or the secondary cache). This takes on the order of 10 cycles, although this number varies widely among different systems. As soon as the word is loaded, the load instruction completes and any instruction(s) waiting for the load to complete can resume execution. Meanwhile, the memory read operation is not complete. The cache always holds contiguous blocks of memory called cache lines. A cache line can range anywhere from 16 to 256 bytes depending upon the particular processor and cache implementation. On the following cycles, one memory word (usually 32 or 64 bits) is loaded at the rate of one word per machine cycle until the entire cache line is filled. This high-speed transfer of multiple words is possible because RAM, in spite of its name, can be accessed sequentially much faster then randomly.

Consider the impact this has on sample block access. If a sample block is not in the cache, the first access to it will fetch an entire cache line of samples. The first access of every cache line will incur a memory delay, but the remaining accesses will take only one cycle because the data will have been prefetched into the cache line. Thus the average sequential access time will be:

$$T = 1 + D/N,$$

Where 1 is the primary cache access time, D is the cache miss penalty incurred by the first access, and N is the number of samples in the cache line. If N and D are approximately equal, then the average access time will be approximately 2 cycles in the worst case. This represents a penalty of only one cycle. It is common to spend many cycles of effort per sample, so an additional cycle per sample is a relatively small overhead. For this reason, we can see small cache effects if we try, but large effects could only arise from a different memory reference pattern. In this light, the shape of the curve in Figure 1 is understandable. We have observed the same behavior in other synthesis systems.

Figure 2 is a dramatic illustration of cache prefetching. The horizontal axis is the phase increment, in samples, of a table-lookup oscillator. The size of the table is purposefully set greater than the size of the cache to generate as many cache misses as possible. The time to generate one sample (in machine cycles) is plotted on the vertical axis. When the phase increment is zero, the same table element is read repeatedly, so there are no cache misses. When the phase increment is 0.5, every sample is fetched twice, so new cache lines are loaded relatively rarely. We expect the rate at which new cache lines are fetched (and therefore average execution time) to increase linearly with the phase increment. This should hold until the phase increment exceeds the line size, at which point every fetch misses the cache and the execution time should level off. This is exactly what we see in Figure 2; actual measurements clearly show that the line size is 32 words (128 bytes) and the cache miss penalty is about 10 cycles.

Memory writes also take advantage of the cache, but these are not so critical because processors usually incorporate write buffers. A write instruction transfers data to the write buffer, and then the data is asynchronously written to memory. Unless total memory bandwidth is a limiting factor (not typical of compute-intensive DSP algorithms), cache misses on memory writes have very little impact on performance.

One caveat is that cache memory sometimes exhibits pathological behavior. If an algorithm is accessing many different memory locations that all map to the same cache line, thrashing may occur as the cache is continually reloaded from different memory locations. This is more typical of direct-mapped caches where each memory address is associated with a unique cache line. VLSI cache implementations have made it economical to implement set-associative primary caches, in which a given memory location can be cached in any of several (typically 4) different cache lines. Although we have not examined the impact of cache design on signal processing performance, we speculate that associative caches virtually eliminate the "hot spot" phenomenon. In practice, we have not observed pathological memory behavior of any consequence.

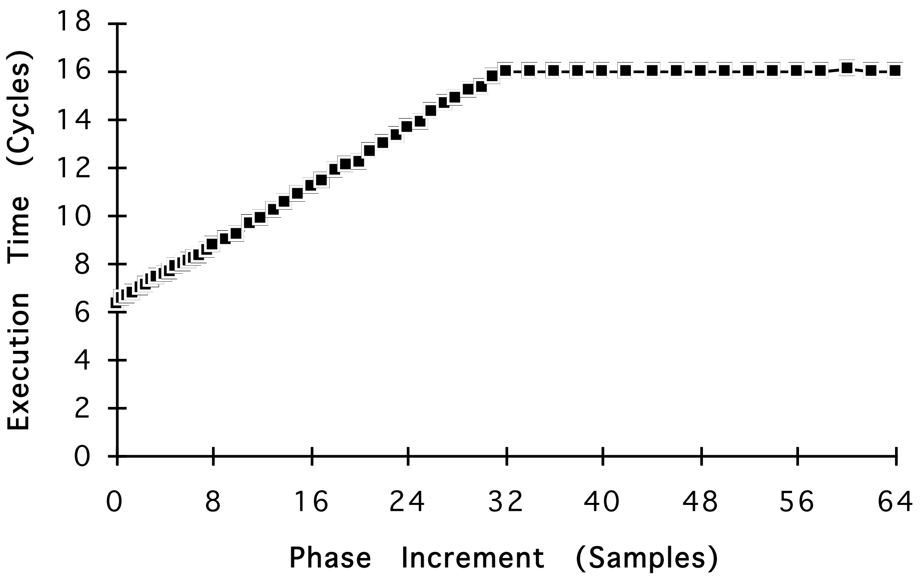

Figure 2. Performance of a table-lookup oscillator as a function of phase increment. Execution time is directly related to the number of fetches per cache line, which in turn is controlled by the phase increment.

Waveform tables are not necessarily read sequentially, so the prefetching behavior of the cache is of limited value. With interpolating oscillators, one or more samples are fetched in sequence, so there is some advantage to prefetching. We will show the effect of table size below in Section "Table Lookup Oscillators."

Inner Loop Size

Is there an optimal inner loop size? We ran experiments in which inner loops were merged and the performance was measured. Merging loops should have at least two effects. First, the loop overhead, if any, is reduced. Second, assuming the first loop was computing and writing samples to be read by the second, the samples can be passed in registers rather than through memory. This eliminates load-store pairs from the computation, removing several instructions and avoiding memory latency.

To study this effect, we compared the traditional system with one in which all the unit generators were combined into one loop. In the former case, intermediate values are stored in memory buffers; in the latter they are stored in local variables. In Figure 3, we plot the improvements of merging loops into one, and passing intermediate results through registers. The benchmark is a series of N first-order filters fed by a table-lookup oscillator. The graph shows performance improvement of a single unit generator over implementing each filter run as a separate unit generator.

One generalization we can make is that larger loops give better performance. We would expect performance to increase as long as all of the data used in the loop fit in the registers. This is what we observe in the case of the PowerPC processor: performance increases until we reach 6 filters, at which point register spilling begins. Even at this point, the compiler begins by spilling read-only filter constants, so the cost of the spill is still less than the cost of storing an intermediate value in memory. With 9 filters the combined loop is still 26% faster than the separate loops.

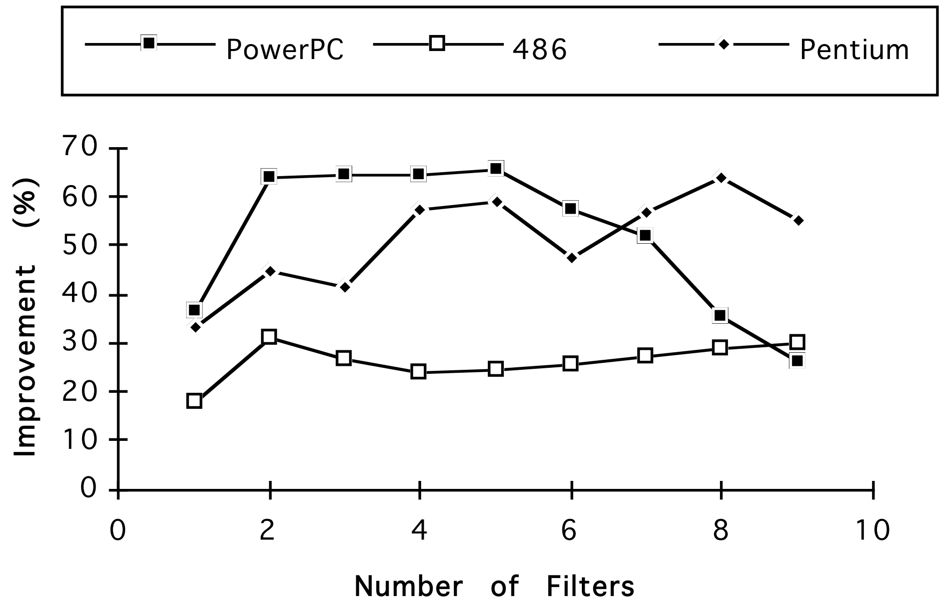

Figure 3. Performance gain is plotted as a function of how many simple filters are merged with a table-lookup oscillator.

On the Pentium, which has very few registers, register spilling is simply a fact of life and is independent of loop size. Overall we see more improvement the more computation is placed inside the loop.

The 486 also shows improvement when unit generators are merged, but since floating point operations dominate the computation time on this processor, the performance gains are less.

Overall, large loop bodies are better, at least to a point. However, making loop bodies larger is a game of diminishing returns. If doubling the size of all loops brings a 50% speedup, doubling again will likely bring only 20%, and so on. Furthermore, if time is already dominated by a few large inner loops, optimizing the small ones will have a small effect.

Control Rate Signals

Control is important, and in any synthetic instrument, it is common to have many envelope generators, low-frequency oscillators, and other control operations. When control signals are computed at audio rates, the control can easily equal the complexity of the audio computation. Alternatively, there is much room for savings if the control information is computed at a lower rate than the audio signal information.

Using low-sample-rate controls is a common practice in both hardware and software systems. Csound is the prime software example. It offers non-interpolated control signals, which are computed at a rate of one control sample per audio block. This means that the control information is conveniently constant for the duration of each inner loop that computes audio.

A problem with the Csound scheme is that control signals, when controlling audio-rate signals, introduce discontinuities at the block boundaries, and this gives rise to audible artifacts. In practice, the block size must be kept small to avoid problems, but this lowers the efficiency of the audio-rate computation. Depending upon the mix of audio-rate and control-rate computation, the control-rate optimization can actually lead to a decrease in overall performance.

Alternative schemes are possible. We studied the following schemes (the numbers correspond to entries in Table 1, which is described later):

1. Incorporate linear interpolation (or a low-pass filter) into unit generators to make control-rate signals smoother.
2. Use non-interpolated control signals as described above.
3. Compute everything at the audio rate.
4. Allow multiple control samples per audio block. For example use block sizes of 64 audio samples and 8 control samples. Control samples are not interpolated, but a new control sample is read for each 8 audio samples. Unit generators that perform mixed computation on control and audio signals consist of two nested loops. The outer loop runs at the control rate and the inner loop runs at the audio rate.
5. Add linear interpolation unit generators to up-sample the control-rate signals to the audio rate. This is similar to method 1 except that here, separate unit generators are used.

We implemented a spectral interpolation synthesis (Serra, Rubine, and Dannenberg 1990) instrument as a benchmark for these 5 approaches. The benchmark, shown in Figure 4, implements two table lookup oscillators with shared frequency controls including a frequency envelope and a vibrato oscillator, which in turn has frequency and modulation depth envelopes. The oscillators are routed to stereo outputs using pan, volume, and cross-fade envelopes. For simplicity, we did not try to share phase computation between the oscillators, and we omitted any wavetable generation code. The result has eleven control unit generators (top of figure 4), six audio unit generators (bottom of figure 4), and in some cases, five interpolators (not illustrated).

Pan unit generators are used to generate attenuation controls that pan between oscillators and between left and right stereo output buses. The inputs to the topmost pan unit generator are a volume x and a stereo pan value y, both of which range from 0 to 1, and the outputs are two control signals, xy and $x(1-y)$. The lower pan unit generators are used to interpolate between the two oscillators. Since signal levels are all determined by control signals, the FM oscillators do not have their own amplitude inputs.

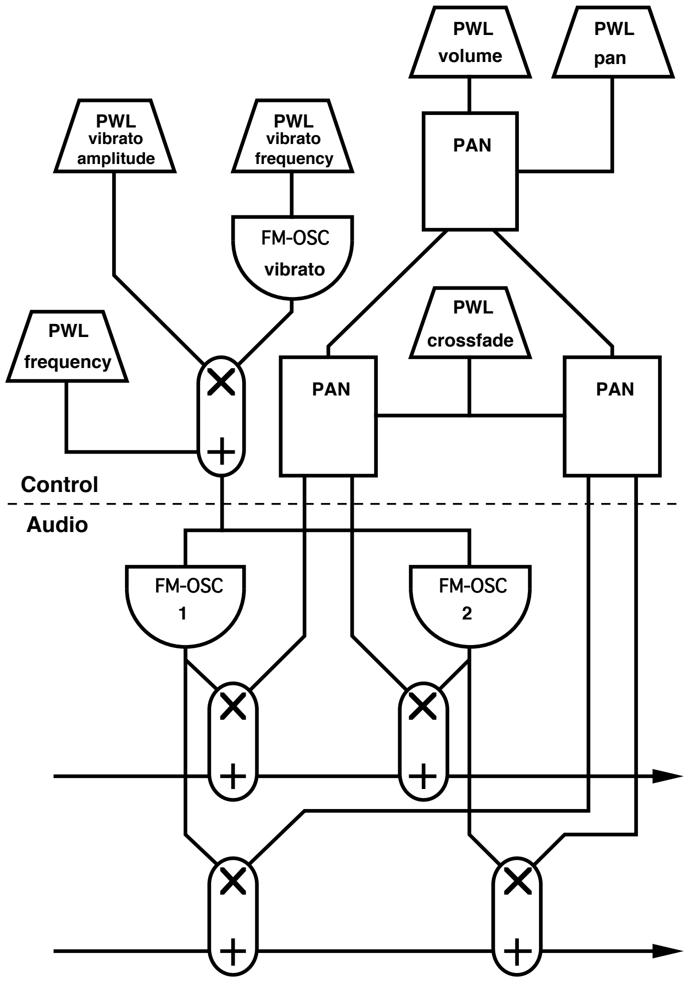

Figure 4. The benchmark synthesis algorithm. PWL (piece-wise linear) generators create control envelope functions. Two FM oscillators share a pitch control. Pan unit generators produce attenuation controls that pan between oscillators and between left and right stereo output buses.

In the first set of tests, we hypothesized that loop setup time would be expensive compared to sample computation time, so we used block computation for both control rate and audio rate unit generators. This hypothesis is examined below.

Run-time was measured by synthesizing the equivalent of 100 seconds of audio as fast as possible. The run-time includes the initialization of the right and left output buffers, so an initialization-only version of the program was also measured and its run-time (1.6s) was subtracted to obtain actual synthesis times. These resulting times, measured on a 25MHz RS/6000 are shown in Table 1.

Table 1. Benchmark execution time under different schemes for handling control signals.

Test Description	*Control Block Size (samples)*	*Audio Block Size (samples)*	*Time (sec)*
1. Linear interpolation	32	32	14.7
2a. Noninterpolated	1	8	17.7
2b. Noninterpolated	1	4	23.2
3. All-audio-rate computation	N/A	32	28.0
4. Noninterpolated, large blocks	32	4 x 8	16.2
5. Interpolating unit generators	32	32	17.5

As shown in the table, the best computation time uses interpolated control signals with audio and control block sizes of 32 samples each, and an audio rate 32 times the control rate. This is better than the Csound approach with an audio block size of 8 (although we have no basis to claim that control signals interpolated over 32 audio samples are equivalent to non-interpolated control signals computed every 8 samples), and substantially better than the Csound approach with a block size of 4. The all-audio rate approach is slower still. It avoids interpolation, but computes nearly twice as many samples, and it is about half as fast as the best method. About 10% slower than the interpolated approach is the non-interpolated approach where audio is computed in 4 sub-blocks of 8 and the audio rate is 8 times the control rate. This does exactly the same computation as the "Non-interpolated" method. Fetching control rate samples 4 times within each unit generator is more efficient that calling the unit generator 4 times as often.

Given that interpolation of control signals is a good strategy, there remains the question of where to perform interpolation. Interpolation can be folded into the inner loops of unit generators, but this requires multiple versions of most unit generators. For example, one version of a multiplying unit generator must multiply a control-rate signal by an audio-rate signal, and another version must multiply two audio-rate signals. Alternatively, interpolation can be done in separate unit generators, eliminating the need for multiple versions of other unit generators. In this benchmark, combining interpolation with other inner loops saves about 20% over the use of separate interpolation unit generators.

In most of these tests, we assumed that control rate signals could be computed in blocks just like the audio rate signals. Does this really save anything? If so, would even larger blocks be better? We found that larger blocks in the all-audio-rate case could give about a 3% to 4% speedup. This means that the implementation, on average, spends about as much time setting up a loop as it does executing the loop one time. If there is more overhead invoking unit generators (as in Nyquist), larger blocks have a greater payoff.

What is the savings due to computing control-rate signals in blocks? As a simplification, let us assume that invoking a unit generator costs the same as computing a sample and that all unit generators have the same per-sample cost. Consider the optimal method in our benchmark, which has 11 control-rate unit generators and 6 audio-rate unit generators. The total cost of computing control-rate samples (in arbitrary units) without blocking is:

$$11 \times (setup + compute) = 11 \times (1 + 1) = 22$$

With blocks of length 32, the average cost per sample is:

$$11 \times (setup + compute) / 32 = 11 \times (1 + 32) / 32 = 11.3$$

In either case, the audio computation per block is:

$$6 \times (1 + 32) = 198$$

Thus, the performance ratio of unblocked to blocked control signals is:

$$(22 + 198) / (11.3 + 198) = 1.05$$

With large ratios of audio rate to control rate like this, we see that computing control-rate signals in blocks only improves performance by about 5%. This is comforting, because in real-time implementations, blocks of low-sample-rate data would impose a long latency.

Algorithmic Improvements

By now it should be obvious that most of the computation time in a software synthesis system will be in the inner loops of the audio-rate unit generators. Anything that will reduce this computation time will have a large impact on the overall performance. In general, we have taken a “purist” approach, insisting on floating point computation and legible, portable, machine-independent code. However, we have encountered a few “tricks” to improve the performance of our inner loops. Whether or not one expends energy in this direction, it is useful to know what sorts of payoffs might be obtained.

It is a natural coding style to compute things sequentially. For example, in an interpolating oscillator inner loop, one might first increment the phase, then perform phase wrapping, then do a table lookup, then interpolate, and finally multiply by an envelope. Since each one of these steps depends upon the previous one, execution will tend to be sequential, even though the processor has the potential for instruction-level parallelism. One optimization technique is to reorder instructions to maximize instruction-level parallelism.

An improvement is to start the table lookup before the phase increment. At lease some of the phase increment can compute while the lookup is in progress. The interpolation computation can be started before the second table lookup, etc. We found that with the RS/6000 running AIX, instructions could be reordered by rearranging statements in the C source code. By first breaking up expressions into sequences of individual operations and then carefully interleaving the operations from different expressions, dramatic improvements can be made. Applying this technique to an FM oscillator, we reduced the cost from 51 cycles to 40, giving a 27.5% speedup. It is important, however, to measure performance. Not all of our attempts at optimization were successful.

When loop bodies are short, loops can be “unrolled,” meaning that the loop body is repeated two or more times. Once the loop is unrolled, there are more candidates for instruction interleaving. Experimenting with an interpolating FM Oscillator (already a large loop body), we were unable to get any advantage from loop unrolling. Furthermore, the process is so error-prone and the resulting code is so unreadable, that we cannot recommend this process unless the payoff of saving a few cycles is extremely high. This is a job for optimizing compilers.

One trick is especially useful for table lookup. The problem here is to convert a floating-point value into an integer table offset. At some point, there must be a floating-point to integer conversion, and this may expand to many instructions or even compile to a subroutine call. Sometimes it is faster to do the conversion directly. For example, the RS/6000 architecture has no float-to-integer instruction. The Pentium, which has a conversion instruction, must use extra instructions to achieve C truncation semantics.

The trick is to use floating-point normalization to place the desired bits in the desired location. If we add 2^{52} (in IEEE standard double-precision format) to a small positive 64-bit floating-point number X, it will be shifted right to align its bits with those of 2^{52}. This normalization process will shift the fractional bits of X "off the end" of the 64-bit representation, leaving the integer part in the low 32 bits of the double word. Thus, a fast rounding algorithm is: add 2^{52}, store as a 64-bit float to memory, read the second word as a 32-bit integer. The add, store, and load instructions can be interleaved with other computations to hide the latency of these instructions.

In an elaboration of this same idea, we can optimize phase computation, where a phase accumulator must "wrap around" when it exceeds the size of a lookup table. Here, we want to retain some fractional bits of the phase. By adding 2^{20} instead of 2^{52}, we can retain 32 fractional phase bits. Assuming the table size is a power of two, we can accomplish phase wrapping by carefully masking the high order 32 bits, taking care to preserve the exponent (use "logical and" to clear bits, and "logical or" to set them). Note that 2^{20} is somewhat arbitrary, and there are many variants of this technique.

Combining these truncation ideas saves 24 of 75 cycles in the inner loop of an FM oscillator on an RS/6000. We applied the instruction reordering optimization described above to save another 11 cycles. The resulting total speedup of nearly 90% illustrates the gap between loops coded in a straightforward fashion and loops optimized for performance.

Table Lookup Oscillators

After looking at various ways to optimize performance, we used the best strategy (linearly interpolated control signals), combined with all of the optimizations described for oscillator implementation, and examined the effect of table size on performance. Figure 5 shows RS/6000 execution time as a function of the table size (in floating point words). For now, consider only the curve labeled "Normal." A total of three tables were used: 2 at the audio rate and one at the control rate. Since the control rate is a factor of 32 less than the audio rate, it is probably best to disregard the control rate oscillator and think of the total table data in bytes as approximately eight times the given table size (four bytes per sample times two tables).

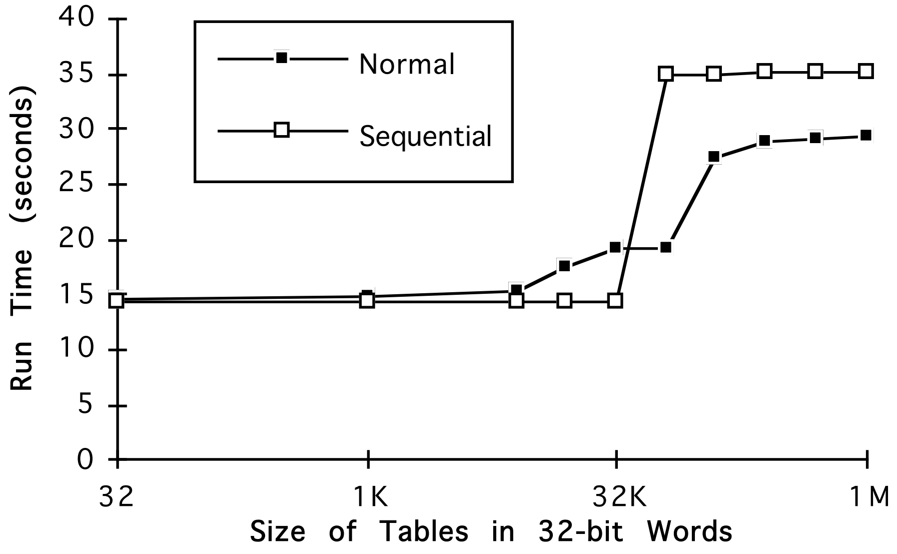

Figure 5. Benchmark execution time as a function of table size. Access to tables is either "Normal" as necessary to compute a roughly 440Hz tone, or "Sequential," for which the oscillator artificially reads memory words consecutively.

As shown in the table, performance is roughly constant out to about 8K (64KB). After that, the performance drops to about half at 128K (1MB), at which point it seems to level off. Using the numbers available, we can estimate the cost of the table-lookup operations with 1M-word tables as 1.72µs, or about 43 machine cycles. Even though each oscillator makes two sequential memory reads, we assume that the second word will be prefetched into the cache, so all of the cost is due to the first read. Note also that the actual read latency will be less than these computed numbers because some of the additional measured cost is due to the need to reload displaced cache lines. Overall, we can say that a random memory read costs about 43 machine cycles.

Note that the performance curve has two "knees." With 64K-word tables (512KB), the effective random access memory read time is about 0.5µs, or about 13 cycles. We suspect that this reflects the cost of cache misses, while the 43 cycle figure reflects the additional cost of reloading virtual address translation registers.

Virtual address translation is yet another source of memory latency. The RS/6000 uses a cache of 128 entries (known as the translation look-aside buffer, or TLB) to translate virtual addresses to real addresses. (Bakoglu, Grohoski, and Montoye 1990) Each entry corresponds to one 4KB page, so the translations for 512KB fit in the cache. When a memory address is referenced but is not in the cache, hardware searches a hash table for the translation and saves it in the TLB.

In our benchmark, we would expect to see performance degradation when the program data exceeds 512KB, corresponding to two tables of 64K-words. The "Normal" curve begins to degrade shortly after this point.

In practice, tables will be much smaller, but for this synthesis algorithm, there are 2 active tables per voice. For interpolating oscillators, 512 samples (2KB) is a small but reasonable choice. Up to 32 tables (16 voices) could be accommodated before the first knee, and this should be realizable at 44.1K samples per second with 75M computation cycles per second. The current crop of PowerPC processors, running at 100 MHz and up, should have enough cycles left over for control, audio output, and wavetable generation.

At the base of the second knee, 256 tables (128 voices) could be accommodated. This would require about 750M cycles per second, which is in the range of expected processor performance at the end of the decade. By then, we can expect larger caches but a greater relative cache miss penalty, so it is difficult to predict performance. These numbers are encouraging in any case.

In our measurements with large tables, each sample was computed from a table location that was far from the previous one. This may not correspond to practice. For example, sample-playback synthesizers usually store samples at or near the playback sample rate, which implies that samples will be read almost sequentially. This should make for excellent cache performance. To test this hypothesis, we modified our oscillators to perform all the usual computation, but then ignore the computed phase and access the next sample in the table.

As shown in Figure 5 (the "Sequential" curve), large sequentially read tables do not cause any performance degradation until the 64K-word tables (representing 512KB of data). At this point, corresponding to the size of the TLB, there is a dramatic increase in cost. Given sequential memory access, we would expect TLB misses to be amortized across many accesses and not impact the computation significantly, so the performance degradation is surprising. No paging occurred in any of these tests.

Conclusions

After considerable benchmarking and testing, we feel that we have a good understanding of software synthesis and how to implement it efficiently on a modern processor. The following recommendations can be made:

1. Audio sample-rate computations should be performed in blocks. Blocks should be large relative to the time it takes to invoke an operation and initialize the inner loop. Because of the prefetching behavior of caches, it is not important that blocks fit in the cache.

2. Control computations should be performed at low sample rates if possible. Since low-sample-rate signals sometimes lead to artifacts such as "zipper noise," linear interpolation should be used. (Note, however, that this adds to latency since interpolation requires one-sample look-ahead.)

3. Unit generator inner loops should be larger rather than smaller. Incorporating control signal interpolation into the unit generator loop is a good optimization.

4. In extreme cases, a special compiler or code generator could be used to construct special unit generators with extended inner loops.

5. Control-rate computations should not be computed in blocks. Non-blocked control signals will minimize the latency in real-time systems, the cost is small, and the implementation is simplified.

We note that cache performance seems to be quite predictable. Three potential trouble spots seem to be: waveform tables, large amounts of sample memory, and multitasking. Up to a point, accessing waveform memory imposes an extra cost because memory is not in the cache. The cost however, is easy to measure. When large amounts of sample memory are accessed, even sequentially, there is the risk of extra cost associated with the loading of virtual memory translation registers. Whether or not this will happen is a function of how much memory is referenced. Again, it is easy to measure the cost and the point at which degradation will occur. Given that real-time synthesis systems do not take advantage of virtual memory, it is too bad that virtual address translation is potentially so expensive.

Finally, it is often assumed that with interrupts and context switching, it is impossible to say what real cache performance will be obtained, and therefore one cannot predict signal processing performance. We like to think of the cost of an interrupt as its execution time *plus* the time it will take to reload the cache when signal processing resumes. Both of these numbers are fairly deterministic and measurable. Having worked with real-time computer music systems with less than 1 MIP overall, we find the idea of “only” 10 MIPS left over for interrupt processing quite attractive.

Most of our measurements were performed on an RS/6000. In limited tests with a PowerPC 601 chip and a Pentium, the results seem to carry over, and in the PowerPC case, performance scales linearly with clock speed. All this is encouraging, but the reader is warned not to assume too much and to measure the performance parameters of any particular machine. We hope the observations and techniques described in this paper will contribute to a better understanding, design, and performance characterization of software synthesis systems.

Acknowledgements

The authors would like to thank IBM and Intel for machines on which to perform this research. Discussions with Fred Gustavson and colleagues at IBM T. J. Watson Research Center were very helpful in setting directions for this work. Thanks are also due to the IBM Computer Music Center for helpful discussions, contacts, and information.

References

Agarwal, R. C., F. G. Gustavson, and M. Zubair. 1994. “Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms.” In *IBM Journal of Research and Development* 38(4) (September): 563-76.

Bakoglu, H. B., G. F. Grohoski, and R. K. Montoye 1990. The IBM RISC System/6000 processor: Hardware Overview. In *IBM Journal of Research and Development* 34(1): 12-22.

Dannenberg, R. B. 1997. "Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis," *Computer Music Journal*, 21(3):50-60.

Dannenberg, R. B., P. McAvinney, and D. Rubine 1986. "Arctic: A Functional Language for Real- Time Systems." *Computer Music Journal* 10(4):67-78.

Dannenberg, R. B., and C. W. Mercer. 1992. "Real-Time Software Synthesis on Superscalar Architectures." In *Proceedings of the 1992 International Computer Music Conference*. International Computer Music Association. pp. 174-177.

Freed, A. 1993. "Guidelines for Signal Processing Applications in C." *C Users Journal* 11(9) (September): 85-93, .

Freed, A. 1994. "Codevelopment of User Interface, Control, and Digital Signal Processing with the HTM Environment." In*Proceedings of the International Conference on Signal Processing Applications and Technology,*. San Francisco: Miller Freeman, Inc. (Also online at http://www.cnmat.berkeley.edu/~adrian/HTM/ICSPAT1994.html.)

Lindemann, E., F. Dechelle, B. Smith, and M. Starkier. 1991. "The Architecture of the IRCAM Musical Workstation." *Computer Music Journal* 15(3) (fall): 41-49.

Serra, M.-H., D. Rubine, and R. B. Dannenberg. 1990. "Analysis and Synthesis of Tones by Spectral Interpolation." *Journal of the Audio Engineering Society* 38(3) (March): 111-128.

Vercoe, B., and D. Ellis. 1990. "Real-Time CSOUND: Software Synthesis with Sensing and Control." In S. Arnold and G. Hair (editors): *ICMC Glasgow 1990 Proceedings*. San Francisco: International Computer Music Association. pp. 209-211.