Real-Time Software Synthesis on Superscalar Architectures[1]

Roger B. Dannenberg and Clifford W. Mercer
Carnegie Mellon University School of Computer Science
Pittsburgh, PA 15213 USA
Email: dannenberg@cs.cmu.edu, mercer@cs.cmu.edu

ABSTRACT

Advances in processor technology will make it possible to use general-purpose personal computers as real-time signal processors. This will enable highly-integrated ‘‘all-software’’ systems for music processing. To this end, the performance of a present generation superscalar processor running synthesis software is measured and analyzed. A real-time reimplementation of Fugue, now called Nyquist, takes advantage of the superscalar synthesis approach, integrating symbolic and signal processing. Performance of Nyquist is compared to Csound.

1. 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. We are developing a real-time implementation of the composition and synthesis language Nyquist for these future processors.

Nyquist embraces a very high-level synthesis model, where entire sounds are values which may be passed as function parameters and returned as results. Nyquist is attractive to composers, but difficult to implement. For example, infinite duration sounds offer flexibility to composers, but require special implementation support. When multiplied by a finite envelope, an infinite sound computation is terminated and garbage collected to avoid wasteful computation. To allow infinite signals and to run in real time, signal computation must be incremental.

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 the Nyquist implementation and how these requirements are met. And finally, we compare the performance of Nyquist with the performance of Csound and a DSP chip on identical tasks.

2. Superscalar Architecture and Nyquist

Superscalar processors represent the state of the art in computer architecture. Current examples include the Intel i860 and IBM RS/6000 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.

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 91] 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. Vercoe’s Csound [Vercoe 90] running on a DEC workstation is a better illustration of the ‘‘all software’’ approach we believe will soon be the

---

[1]Published as: Dannenberg and Mercer, ‘‘Real-Time Software Synthesis on Superscalar Architectures,’’ in ***Proceedings of the 1992 International Computer Music Conference***, International Computer Music Association, (October 1992), pp. 174-177.

norm. Even without the benefits of a superscalar processor and a real-time operating system, Csound illustrates impressive performance. Our work in this area began in 1983 with the design of Arctic [Dannenberg 86], 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 is based on Fugue [Dannenberg 91a] which in turn 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. We set out to answer the following questions: What characteristics of superscalars are important for music synthesis? What new techniques are necessary to execute Nyquist in real time? What is the overhead or benefit of the advanced features of Nyquist?

Benchmark. For our study, we used a 30MHz IBM RS/6000 Model 530 running AIX; all code was written in C and compiled with optimization. Our benchmark is the generation of a sequence of 40 tones, each of which has 12 partials of constant frequency and piece-wise linear amplitude envelopes. The tones are sampled at 44100Hz and the total sound duration is 14.4 seconds. The sound samples are discarded as they are computed to avoid I/O, and we measured total real computation time. Our study used both Csound and Nyquist.

3. Optimizing for Superscalars

We first consider several characteristics of superscalars that might bear on implementation strategies.

Are blocks important? Many DSP systems assemble unit generators into a monolithic loop, each iteration of which generates one sample. In contrast, most software systems amortize the overhead of loading parameters into registers and caching instructions by computing samples in blocks.

Not surprisingly, block computations give dramatic speedup: Csound (which has the lower overhead per block) improves by a factor of about 7 with large blocks (see Figure 1).[2]

Is caching critical to performance? If so, we would expect small block sizes (relative to the cache) to be important. Again using Csound, we do not see any significant speedup with small block sizes. This indicates that any data cache improvement is offset by instruction cache misses and parameter setup overhead.

Does arithmetic dominate computation time? Consider the problem of multiplying an audio-rate signal A by a control rate signal B, where B must be linearly interpolated (see Figure 2). The interpolation could be done a block at a time before the multiplication, or the interpolation could be merged into the inner loop of the multiplication. On machines with slow floating point operations, we expected and observed little difference between the two examples. With a superscalar, we expected the integrated interpolation to run faster because there is less loop overhead and fewer memory loads and stores. Surprisingly, there is only 6% speedup in the integrated case. However, we discovered a compiler-generated procedure call (floating point to integer conversion) required for interpolation. The moral is: Peak performance is very hard to achieve.

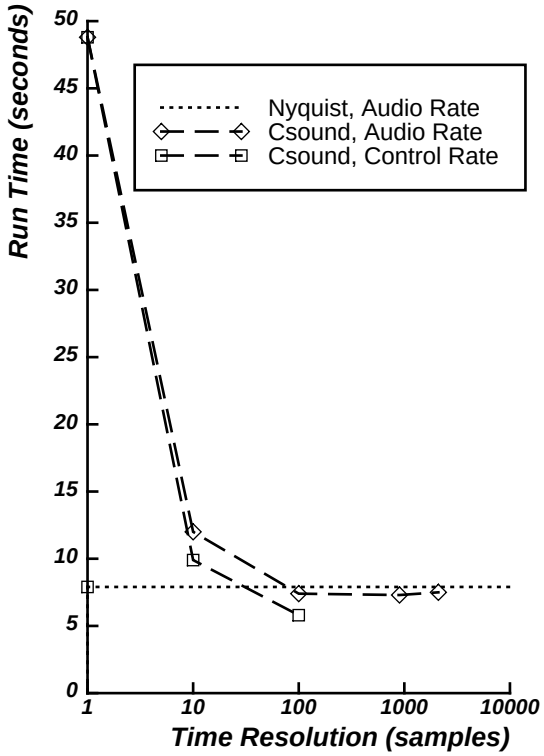
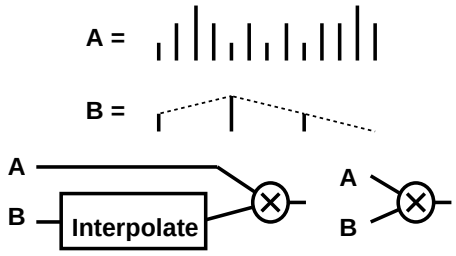


Figure 1: Performance of Nyquist and Csound as a function of time resolution (or Csound block size). Csound rounds envelope breakpoints (both audio and control rate) to the nearest block boundary. Nyquist uses variable sized blocks, so timing resolution is always 1 sample. See Section 5.

In another experiment, we wrote two versions of a Csound orchestra such that one version performed additional loads from memory. We determined that a single load costs 220ns, which accounts for about 23% of the total synthesis time. On the other hand, we split the Nyquist amplitude-modulated sinusoid generator into a sinusoid generator (without AM) followed by a multiply unit generator (implying an additional store and another loop iteration)

[2]This is a bit misleading, however, because we did not rewrite Csound for the special case of blocksize = 1.



	Separate Interpolation	Integrated Interpolation	Time Ratio
68000	720	710	1.01
68040	5.6	4.9	1.14
Sparc	2.12	1.81	1.17
RS/6K	1.68	1.59	1.06

Figure 2: When multiplying signals with different sample rates, the interpolation can be performed before the multiplication (left) or as part of a more complex multiplication operation (right). Times for 68000 (software floating point, 8MHz), Sun Sparc IPX (RISC with floating point, 40MHz), NeXT 68040 (CISC, 25MHz), and IBM RS/6000 (superscalar, 30MHz) processors are shown in microseconds per output sample.

and the cost was only 170ns, or 16%. Thus, merging unit generators to eliminate loads, stores, and other overhead has the potential to provide at least moderate speedup. Because of parallel and pipelined execution, however, the effect of restructuring a computation is hard to predict.

4. Real-Time Nyquist

We now turn to new implementation techniques. Originally, Fugue computed signals by allocating a single block of memory and computing samples for the entire duration of each signal. Real-time computation, however, dictates that signals be computed incrementally instead of *in toto*. The new implementation (Nyquist) computes only the signals and combining operations that are necessary to produce the sound for the next interval in time. This amounts to a restructuring of the schedule of sound computation. If we consider the whole composition as a tree, we now schedule the computation of sounds in a breadth-first manner instead of scheduling the computation in depth-first order. The requirement for real-time synthesis is that the time to compute each interval of sound is less than the duration of the interval itself.

This rescheduling of the computation of sounds has other implications in addition to the incremental availability of the final output sound. The memory usage of Nyquist is smooth instead of bursty. Since Nyquist computes the signals incrementally, only the next interval of sound need be extant in memory. The memory that was used earlier in the signal may be reclaimed, and the future samples need not be produced and stored in memory until they are needed. This is in contrast to the Fugue implementation which allocated large buffers to hold entire signals as they were being computed. In short, the incremental evaluation implies a block-based memory management approach for computing signals.

Implementation. Space limitations allow only a brief description of the Nyquist implementation. Figure 3 illustrates that a sound consists of a list of blocks of samples. The list terminates at a suspension object, which can be requested to extend the list, computing a new block. Suspensions typically store pointers to other sounds as shown. Sounds are accessed via headers which store the sample rate, scale factor, and various other parameters. A sound can be shared, e.g. in Figure 3, variables A and B reference the same sound via separate headers. Sounds can only be accessed sequentially; here, B has read more samples than A. Sound list nodes and sample blocks are reference counted. When the head of a list has no more references from sound headers, it is freed for reuse. Sound headers are garbage collected as part of the normal Lisp memory management system.

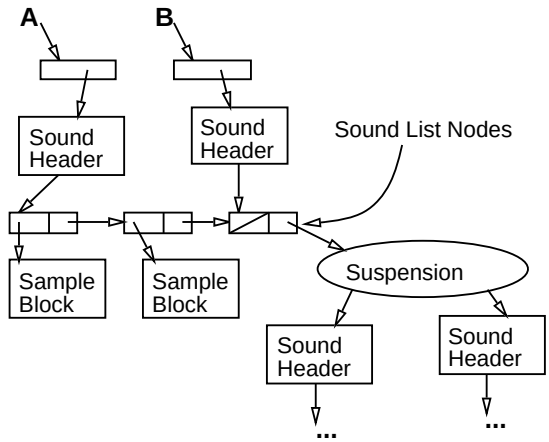


Figure 3: Run-time data structures for Nyquist.

The most interesting operators in Nyquist are **`seq`** and **`s-add`**. The **`s-add`** operator adds two sounds, and is optimized to deal with the case where the sounds do not start and stop concurrently.

Consider the expression **`(s-add A B)`**, where **`A`** starts before **`B`**. By definition, sounds are zero before their start times, but it would be inefficient to add **`A`** to zero until **`B`** starts. Even copying samples from **`A`** to the result could be inefficient. The **`s-add`** operator solves the problem by copying only pointers to sample blocks. This is the reason sound list nodes are separate from sample blocks: A sample block can be referenced from many sound lists.

The **`s-add`** operator allows us to efficiently add a sequence of thousands of notes, but this requires thousands

of sounds to be created at the beginning. An alternative is to use **seq**, which defers creation of the second sound until the first one stops. The subsequent sound is represented as a Lisp expression (usually a closure), which is not evaluated until the sound is needed. This provides a representation that is both time- and space-efficient.

5. Performance Evaluation

Nyquist is surprisingly efficient. With a block size of 1024, Nyquist spends about 92% of its time in inner loops. Optimization outside of the loops should be possible, allowing shorter blocks for low-latency real-time applications. Since the inner loops are identical to those of Csound, Nyquist has nearly the performance of Csound as long as block sizes are large. (Csound is about 20% faster.)

However, large block sizes in Csound produce audible distortion when control rate signals are used. Even when audio rate signals are used throughout, notes must always start on block boundaries. Thus, with a block size of 100, Csound at 44.1KHz quantizes times to about 2ms. Nyquist, on the other hand, uses variable length blocks and quantizes times to the audio sample rate, i.e. about 23us, corresponding to a Csound block size of 1. If quantization is to be avoided, Nyquist is about 6 times faster than Csound.

Instead of looking at extremes, let us consider typical parameters. A typical use of Csound might be to run with a control rate that is one tenth (0.1) of the audio rate in order to limit distortion. Nyquist can perform the same benchmark computation entirely at the audio rate and still be 20% faster[3].

Comparison with DSP's is difficult due to the difference in program structure and functionality, but after adjusting for clock rate differences, the hand-microcoded Kyma system [Scaletti 91] and NeXT sound kit [Jaffe 89] run our particular benchmark on a single M56001 at most 3 times faster than Nyquist running on an RS/6000. It would be interesting to determine how much of this difference is due to machine architecture, to floating-point, to hand optimization, and to the structure of Nyquist.

[3]One might argue that the benchmark penalizes Csound by having only a few simple control-rate envelopes: with enough control rate signals, Csound would probably win out, but if Nyquist is also allowed to compute at control rates (a feature of Nyquist), Nyquist's advantage will actually widen because even Nyquist's control-rate signals are processed in blocks.

6. Conclusions

Superscalar processors show great promise for music audio processing. We found that a RS/6000 programmed in C performs floating point DSP about a third as fast as an M56001 chip running hand-microcoded integer DSP. Superscalar performance will increase rapidly and our code will be easy to port.

Given the high degree of pipelining, a cache, and very fast floating point on the RS/6000, we expected to see the sort of performance unpredictability that drives people to program DSPs. The effects we observed are surprisingly small.

Nyquist is a high-level language that illustrates the advantages of integrated symbolic and signal processing made possible by superscalar processors. We are quite pleased to see that Nyquist can offer greater flexibility and precision than conventional sound synthesis systems without any significant loss in performance.

In the future, we plan to extend Nyquist's set of unit generators to make it more generally useful as a non-real-time system. We need to study interpolation strategies for efficient mixed-sample-rate computations. For real-time applications, we need to address the garbage collection issue, perhaps by replacing our Lisp interpreter. We also plan to explore the resource-instance model [Dannenberg 91b] in this context.

7. Acknowledgments

This work was supported in part by MicroMed Systems as part of NSF grant SBIR ISI 9000272 and by an NSF Graduate Fellowship. The authors would also like to thank Barry Vercoe for making Csound available. We learned from his excellent and efficient code, and it also turned out to be a very useful tool for testing various performance hypotheses. The authors also wish to thank Carla Scaletti and Julius Smith for M56001 performance measurements, Dean Rubine for many insights, Joe Newcomer for helping with Nyquist implementation, and Peter Velikonja for Csound consultation.

References

[Dannenberg 86] Dannenberg, R. B., P. McAvinney, and D. Rubine. Arctic: A Functional Language for Real-Time Systems. *Computer Music Journal* 10(4):67-78, Winter, 1986.

[Dannenberg 91a] Dannenberg, R. B., C. L. Fraley, and P. Velikonja. Fugue: A Functional Language for Sound Synthesis. *Computer* 24(7):36-42, July, 1991.

[Dannenberg 91b] Dannenberg, R. B., D. Rubine, T. Neuendorffer. The Resource-Instance Model of Music Representation. In B. Alphonse and B. Pennycook (editor), *ICMC Montreal 1991 Proceedings*, pages 428-432. International Computer Music Association, San Francisco, 1991.

[Jaffe 89] Jaffe, D., and L. Boynton. An Overview of the Sound and Music Kit for the NeXT Computer. *Computer Music Journal* 13(2):48-55, 1989.

[Lindemann 91] Lindemann, E., F. Dechelle, B. Smith, and M. Starkier. The Architecture of the IRCAM Musical Workstation. *Computer Music Journal* 15(3):41-49, Fall, 1991.

[Scaletti 91] Scaletti, C., and K. Hebel. An Object-based Representation for Digital Audio Signals. *Representations of Musical Signals.* In G. De Poli, A. Piccialli, and C. Roads, MIT Press, Cambridge, Mass., 1991, pages 371-389, Chapter 11.

[Vercoe 90] Vercoe, B. and D. Ellis. Real-Time CSOUND: Software Synthesis with Sensing and Control. In S. Arnold and G. Hair (editor), *ICMC Glasgow 1990 Proceedings*, pages 209-211. International Computer Music Association, 1990.