Optimizing Software Synthesis Performance*

Nicholas Thompson and Roger B. Dannenberg
School of Computer Science, Carnegie-Mellon University
nix@cs.cmu.edu, rbd@cs.cmu.edu

Abstract

Sound synthesis in software on general-purpose computers offers a number of advantages over specialized hardware. Its main disadvantage is performance; however, as general purpose machines get faster, software synthesis becomes increasingly accessible. The CPUs used in the latest generation of personal computers achieve high performance using large amounts of internal parallelism in the forms of multiple functional units and deep pipelines. These performance improvements in the CPU do not, in general, carry over to the performance of the memory system, changing the importance of different optimizations. We report on several profiling experiments which suggest optimizations for these processors.

Introduction

Our work aims to determine what factors affect the performance of software synthesis (Dannenberg), what is the optimal structure for synthesis software, and how much room exists for improvement. We divide the work done by a software synthesizer into three parts: useful work (the computational core of the routine), communication costs (including memory access and cache miss costs), and loop overhead (essentially constant for each unit generator). We treat these parts separately below.

Computational Improvements

In the core of the routines, we have found a technique for phase rounding and wrapping that yields a performance increase of nearly 50 percent in our FM oscillator code. Our measurements were taken on an IBM RS/6000 model 250. however, the rounding technique relies only on IEEE arithmetic. Rounding is accomplished by adding 2^{52} to the double-precision phase, which forces the floating-point unit to shift the mantissa in such a way as to make the exponent 0. This allows the integer part of the phase to be easily extracted from the binary representation of the new number. Using this method of rounding rather than the built-in C cast to integer saves 19 (of 75) instruction cycles per sample in the oscillator loop, for a performance increase of 35 percent. A similar technique can be used to implement phase wraparound if the table length is a power of 2: once again the mantissa is shifted by adding a constant, and the high bits of the mantissa are masked out using logical operations. This eliminates 5 more instruction cycles, for a further improvement of 9.6 percent. Finally, by reordering expressions in the loop to expose more instruction-level parallelism, another 11 instruction cycles can be saved, bringing the total cost to 40 instruction cycles per sample for a total performance improvement of 90 percent. It is surprising how much speedup can be obtained by slight variations in code order.

Communication Costs

The traditional way to combine flexibility and efficiency in a software synthesis system is to build separate "unit generators," each of which performs a relatively simple computation (Mathews). Unit generators are then combined to form more complex systems, with communication between unit generators taking place through memory buffers. Efficiency is good because the overhead of starting and stopping unit generators is amortized over the number of samples in the buffer, but the cost of reading and writing the buffers is still incurred for every sample. To test the cost of communication between unit generators, we manually fused groups of unit generators into single loops, using local variables for communication between steps. The possible cost of this approach is that larger loops may require more variables (such as filter coefficients and buffer pointers) to be stored in memory than in registers: if one has more state than registers, one must pay memory costs in any case. We then compare the speed of this fused loop with the speed of the same computation using simple unit generators communicating through buffers in memory.

The particular application chosen is a simple wavetable oscillator whose output is connected to a chain of from 1 to 9 first-order filters. Figure 1 compares the cost of the traditional approach (communication through memory) with the cost of the corresponding monolithic unit generator (communication through registers) on the RS/6000.

*Published as: Nicholas Thompson and Roger B. Dannenberg, "Optimizing Software Synthesis Performance," in *Proceedings of the 1995 International Computer Music Conference,* Banff, Canada, September 1995. International Computer Music Association, 1995. pp. 235-236.

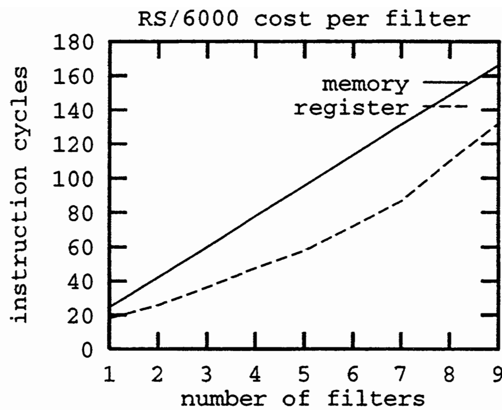

Figure 1

The cost of register spillage is less than expected: even with 9 filters combined into one loop, performance is better than with the separated loops. Inspection of the assembly code for the RS/6000 shows that, although register spillage begins when 6 filters are combined into the loop, the cost is small: read-only filter coefficients are spilled before read-write filter state (so there is no store cost), and there is enough computation to be done that reloading spilled coefficients can proceed in parallel with the rest of the computation.

Figure 2 compares the performance gains on the RS/6000 and Intel 486. Removing communication through memory produces significant speedups on the RS/6000 architecture (with fast floating point and a large register file). The 486, with slow floating point and hardly any registers, shows only a slight improvement, while preliminary Pentium results show speedups similar to those on the RS/6000. This work shows that, as processors incorporate more registers and parallelism, the optimal size for unit generators will rise.

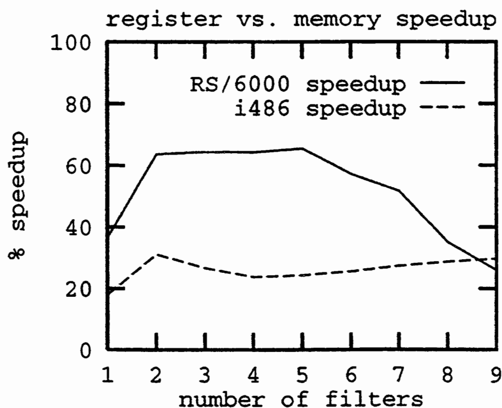

Figure 2

Cache Influences

Conventional wisdom holds that cache misses cause a significant performance penalty on modern architectures, and some software synthesis systems (Lindemann et. al.) have been designed based on this premise. We conducted some simple tests which show that typical software synthesis code is served very well by cache prefetch, so that the cost of a primary cache miss is amortized over as many data items as fit in a cache line. For example, on one Intel 486DX2-50 machine, the cost of a cache miss is 8 instruction cycles. Since we are using single-precision floating point samples (4 bytes each) and the cache line size is 16 bytes, the actual cache miss penalty is only 2 instruction cycles per sample, which is negligible compared to the cost of computation. As with other communication costs, larger loop bodies will be more efficient.

Conclusions

Superscalar processors offer new challenges for optimization of software synthesis code. Much of the performance gain to be had requires careful attention to floating-point operations and instruction-level parallelism. In addition, the cost of using unit generators which communicate through memory has risen significantly and will probably continue to do so. Although we were able to speed up computation by combining unit generators, this is a tedious and error-prone process if done manually. Using this optimization successfully on a large scale will require automatic code generation.

Acknowledgements

The authors would like to thank Fred Gustavson at IBM for showing us the fast rounding trick and stressing the importance of code order.

Bibliography

Mathews, M. V. 1969. *The Technology of Computer Music.* Boston: MIT Press.

Dannenberg, R. B. 1992. "Real-Time Software Synthesis on Superscalar Architectures." *Proceedings of the 1992 ICMC.* San Francisco:International Computer Music Association. pp. 174-177.

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