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

Course Project Milestone Report
CMU CS 740: Computer Architecture, Fall 2002
Prof. Seth Goldstein

Ning Hu (ninghu@cs.cmu.edu) and Vahe Poladian (poladian@cs.cmu.edui)

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

Major Changes

So far the project is in good pace. Most of the results we got are not far different from what we expected, thus there is no major changes.

What We Have Accomplished So Far

- **Things learned**
  - **VTune**: We have learned how to use VTune to do sampling and analysis sessions. The results reveal specific bottlenecks of the application.
  - **Visual Studio Profiler**: We used the profiler to compare the performance between different versions of the application.
  - **SSE**: We have learned its instruction sets and implementation details
  - **Searching the online documentations**: We have searched the Web extensively for documentations that will be helpful for our project. We got a lot of very useful information, e.g. analysis and optimization on floating point to integer conversion operation; SSE code optimization, etc.

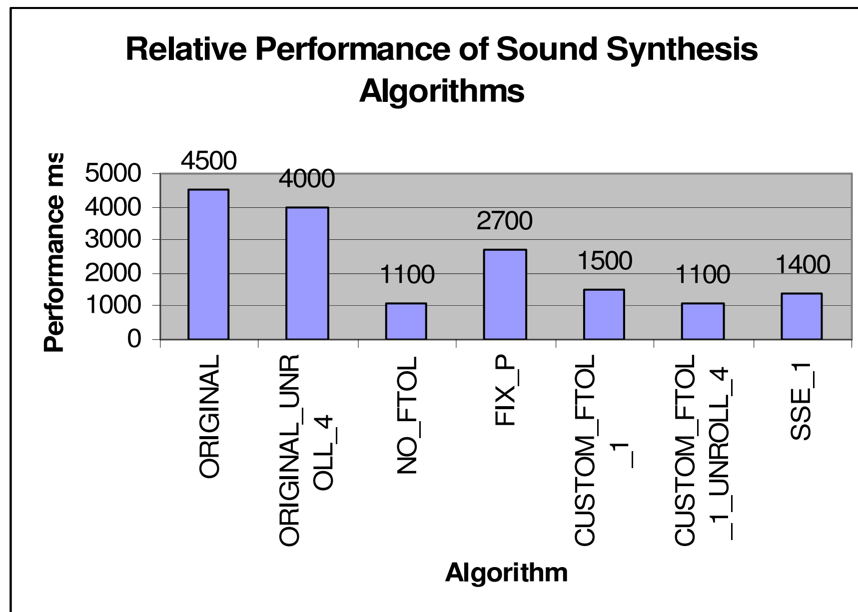
- **Application optimization iterations**
  - We extracted the code we need from the original sound synthesis language Nyquist and use it as the **base version**.
  - We did two initial optimization versions which unrolls the key loop twice and unrolls the loop four times respectively. They have the speedups of 6% and 10% comparing with the base version.
  - We figured out the bottleneck of the base version is the operation that does the floating point to integer conversion (FTOL). Thus we tried out four versions to get away with the bottleneck.
    - Integral-decimal-apart version (NO_FTOL). In this version, the phase variable is represented in the combination of one integer variable that stores the integral part, and one floating point variable that stores the decimal part. It has the speedup of 75%, which outperforms any other versions of the application. But we will not do further optimization on this version as it is too specific to this application.
    - Standard fixed point version (FIX_P). In this version, the phase variable is represented in standard fixed point representation. It has the

speedup of 40%. The reason why it doesn't perform the best is the linear interpolation process needs to get the decimal part from the packed fixed point representation, which requires expensive computation of either multiplication or division.

    - Two versions that use the general floating point truncation optimization (CUSTOM_FTOL 1 & 2). We learned the optimization algorithms from online documentations and implemented in our application. They have similar speedups of about 65%. And we choose the better one to be the **second base version** to do further optimization and SSE implementation.

- The unrolling the loop four times version (CUSTOM_FTOL_UNROLL_4). It is based on CUSTOM_FTOL_1 version and has an additional speed up of 20%. This **Pre-SSE version** is prepared for SSE implementation
- Initial SSE implementation. It is based on the Pre-SSE version and so far its preliminary result doesn't perform as good as the Pre-SSE version, but is still better than the original base version. The reason is that memory load and store operations dominate, as we need to prepare the data to pack into SSE registers. We expect to continue optimizing this version.

Below we present a chart showing the relative performance of the various versions. The benchmark used synthesized 1000 seconds of sound at 44,100 hertz. The measurements were taken on Pentium III, 1GHz CPU, under Windows 2000, using code generated by Microsoft Visual C++ 6.0 compiler. Test with the GCC compiler generated similar data.



- **Compilers comparison**

  We compiled all the versions of the application with both Visual C++ and GCC (running in Cygwin environment) compilers and analyzed the results from both of them. The base version of the program generated by GCC outperforms about 30% than the one generated by Visual C++, as the FTOL operation in GCC is in-lined while in Visual C++ is a function call. But once the expensive FTOL operation is gotten away with, the results from GCC and Visual C++ are similar.

Meeting Our Milestone

We did accomplish our milestone as outlined in our proposal. In fact, as the project progresses smoothly, we might have extra time in the end to apply what we've learned to other typical sound synthesis applications related to our current investigating object, like doing the quadratic interpolation instead of linear interpolation, so that we will have a complete and decent analysis result on similar types of sound synthesis applications.

Surprises

Before starting the project, we were already told that the operation of floating point to integer conversion is expensive, but we didn't expect it to be so expensive that we have to tackle it first before we move on. In our base version of the application, at least 50 CPU 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.

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
fistl -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 Athlons rely on deep pipelines to achieve higher peak performance. So this piece of code reduces the floating point performance of the CPU to level of a non-pipelined FPU. Also another important instruction fistl that does the actual conversion work also requires about 6 CPU cycles to complete.

We implemented several optimized version of the application in order to solve the problem caused by the FTOL operation and they all turned out to be successful to some extent. But we want to keep the solution general enough so that it will be useful for other applications.

Also this might actually be a good news for the SSE implementation of the application, as the SSE instruction sets has the instructions dedicated in floating point to integer conversion.

Revised Schedule

Ning

- Implement the pipeline optimized version and analyze the performance
- Write other typical sound synthesis code (e.g. quadratic interpolation) related to this application for advanced analysis
- Help on optimizing the SSE-enhanced version

Vahe

- Continue optimizing the SSE-enhanced version,
- Run experiments,
- Interpret and iterate over the above 3.

All

- Finalize analysis
- Summarize experiment results
- Write-up report
- Prepare poster

Resources Needed

We have all the necessary resources to complete the project. The development tool Quexal mentioned in the proposal is no longer needed as we figured out writing SSE code directly is not that difficult.