An Interpreted Language for Real Time Signal Processing

Guenter Geiger (ggeiger@i uamtg.upf.es)
Music Technology Group,
Pompeu Fabra University
Barcelona, Spain

Abstract

Real time signal processing for music generation has been affordable for several years now, due to the tremendous increase in the performance of computer systems. Parallel to that development several signal processing engines have been built, which are specifically targeted to real time processing, and hence make it easier for the electronic musician to build custom applications [6] [2]. Additionally, they offer fast prototyping systems for researchers. Unfortunately these system are limited in that they only offer block wise arithmetic.

This paper investigates possibilities and performance of interpreted languages for signal processing tasks, in order to overcome these limitations. As an example, the performance of an implementation of a Forth like language is given.

1 Introduction

Due to performance issues, most existing real time signal processing systems use block arithmetic. They are not specifically fit for research work, as block arithmetic, which in some cases is sufficient for algorithms in the audio domain, definitely is not sufficient when developing algorithms in the spectral domain.

The common way to surpass this limitation until now has been to write dedicated code in a low level language such as C, in order to have access to single data samples. This approach is not always well suited for several reasons.

C is a rather low level and insecure language, programming in C therefore needs a certain amount of experience. Basically the same applies to C++.

Researchers who are able to program in a compiled language, loose too much time in the coding, compilation, testing cycle, i.e. they prefer an interleaved coding/testing approach and therefore develop their algorithms on platforms like Matlab that offer an easier way to change and adjust their algorithms.

Naturally it is desirable that the resulting system is a runnable prototype or even better a finished application. With existing systems this is either not possible, or only by introducing further optimization steps, which could or could not lead to a system that runs in realtime.

The goal is therefore to develop an interpreter with a performance that lies in the same range as a custom “C” implementation for a given algorithm. At the first thought this might look very hard, but we have the advantage of having a very well defined field of application for our interpreter, and can therefore target its structure to our needs.

The basic structure of stack engines makes it possible to write very fast interpreters. This structure is discussed in the first part of the paper.

The system discussed in the second part adds functionality specifically for signal processing and takes into account the goals that should be reached by the interpreter.

Some examples show the system applied to different algorithms in signal processing and their performance in comparison to implementation in C and other programming languages.
2 Implementation Considerations

Interpreted languages generally depend on some virtual machine representation on which the basic commands are executed. In order to make this virtual machine hardware independent, it is commonly based on stack engines. A stack engine, as opposed to register engines, keeps values in a data stack instead of registers. Each command is working with the values that are on top of the stack.

The postfix notation is a natural consequence of this stack based design. The natural order of the commands to perform an addition is:

1. push value1 onto the stack
2. push value2 onto the stack
3. add the two values on top of the stack and replace them with the result of the computation

resulting in a command like “4 5 add”.

The basic operations map directly to forth code, therefore making forth some sort of “assembler” for stack-based engines.

Another example of such a stack engine is the Java virtual machine. Commonly virtual machines offer different services, depending on the concrete needs of the corresponding interpreter.

Knowing the implementation of interpreters we can ask ourselves what makes interpreters slower than compilers.

There are two main reasons why interpreters are slower. First, there is a slowdown caused by the mapping of the virtual machine to the real hardware. A software implementation of a stack on a register based machine always adds additional operations and memory accesses for each instruction. Secondly, the operation scheduling is done in software, adding a constant overhead to every instruction for scheduling to the next instruction.

Both of the problems have been studied and solutions have been proposed, leading in one case to the development of hardware stacks for dedicated forth hardware and hardware implementations of Java engines and on the other hand to different techniques to improve the performance like threading, stack caching and just in time compilation.

2.1 Threading [3, 4]

Different techniques of threading are know, each of them has its own advantages and disadvantages. Subroutine threading adds a subroutine for each interpreter command (This technique is commonly used in existing computer music systems such as Csound and the different MAX slangs). Other threading techniques include indirect threading and direct threading. On modern processors direct threading seems to be the fastest solution. In direct threaded systems, there is a direct jump from one implementation of a command to the next, avoiding calls to subroutines and the more expensive switch statement.

2.2 Stack Caching

Threading addresses the first overhead within interpreters, namely the instruction scheduling. To improve the performance of the software stack machine implementation commonly a technique called “stack caching” is performed. Stack caching holds some elements from the stack within registers of the register based hardware. This method has the drawback, that frequently used operations like push and pop get more expensive. Commonly this method is used with only the “Top of Stack” held in a register. Holding more than one element of the stack in registers adds a certain amount of logistic overhead to the implementation, which therefore not always results in better performance [5].

2.3 Common Interpreters

Commonly used fast interpreters like Ocaml (an ML based object oriented functional language) and Smalltalk use similar approaches to optimize speed. The same concepts can be found in the Java virtual machine interpreter.

The system used for the benchmarks is a Forth like interpreter. It is implemented from scratch, in order to avoid side effects from an integration into an unknown system and to have full control over the optimizations applied. A look on the implementation of the kaffe virtual Java engine revield no such optimizations in interpreted mode.

Commonly it can be said that existing optimizations do not specifically target one concrete use of
the the engine, but they try to improve general performance.

3 An Interpreter for Signal Processing

The first thing that is done when making a scripting language fit for realtime DSP, numerical python, matlab... is adding vector computation commands. This hasn't been done yet for the benchmarks, as the benchmarking target is on a lower, sample based, computation level. Benchmarking different vector computation algorithms isn't in the scope of this paper, the goal of the system is not having an extensive slowdown in doing sample wise computation, so that the possibility of vector computations within the language itself is given.

The initial goal of developing the interpreter was to use it as an extension to already existing Computer Music Systems, specifically an implementation for the "PD" language was made. This goal in mind the engine is lacking several features normally found in programming languages. It uses a very simplified memory model, there is no such thing as garbage collection, object orientation, or even the possibility of function definitions. Whereas this is quite sufficient for implementing small algorithms, another approach has to be taken when the target is a self contained system. One solution would be to apply the discussed techniques to an already existing interpreted language.

Several other techniques can be applied in order to make the language even faster.

3.1 The Floating Point Stack

Applying the known techniques of direct threading and stack caching improved the performance of the system as can be seen in the results section. The implemented engine has a separate integer and floating point stack, implementing a stack caching system for the floating point stack led to a performance loss on the Pentium III processor. Obviously the compiler used (gcc) didn't translate the C-code into the desired assembly instructions, leading to additional overhead when trying to do stack caching with floating point registers.

The solution to this problem, on Intel architectures, lies close at hand. The arithmetic coprocessor is implemented as a stack engine, therefore the mapping from virtual instruction to machine instructions is one-to-one in most cases, making the overhead of the interpreter for floating point calculation only dependent on the scheduling overhead.

3.2 Parsing

The simple structure of the system makes it possible to include some sort of compilation into the text parsing state of the system. Generally the text parsing is already some sort of compilation, which in the first stages only consisted of a direct "token to bytecode" mapping. Most of the algorithms in signal processing consists of looping over arrays of data. Knowing that the body of the loop could be taken, arranged without the scheduling instructions (in direct threading these proved to be 3 instructions at the end of each word), and then used as one single bytecode should further speed up the calculations. Parsing is done in the startup of the system and has proved to be very fast. It has to be considered that this phase is equivalent to both, the compilation phase for java programs, and the JIT compilation performed at runtime.

3.3 Type and Memory Checks

The interpreter described lacks one basic feature which is commonly connected with interpreters. It is inherently insecure, making it possible to access objects on the stack with wrong operations (it is not type save) and even worse it is possible to access memory regions that lead to segmentation faults. In order to avoid any overhead produced by these check during runtime, a two way approach has to be taken.

First, the parser should be implemented in a way to be able to do type checking, but additional type checking at runtime is unavoidable.

In this case all speed assumptions made here would be invalid, because they don't take the type checks into account [1].

There is a way around this, by implementing two different parsers, which would make it possible to parse the code into type and memory save or into
performance versions, similar to the debug and release versions of C programs.

This doesn’t guarantee absolute security, but most of the problems are visible during development with the secure version, and there will still be no speed degradation in the runtime version.

4 Results

Several tests have been performed, in order to compare the speed of the interpreter to a compiled C version of the same algorithm. The algorithms used are:

- **Scaling** Here the loop consists of one floating point operation and the corresponding memory fetch and store operations.
- **Basic Fir Filter** Several fetch/store, multiplications and additions in the loop body.
- **Peak Detection** This includes some flow control changes.

All of the tests were made with different settings. The output is the ratio between the compiled and the interpreted version of the algorithm. It has to be stated, that the implementations in C are not specifically tuned, so they give an expectation of what a programmer would do without tuning the implementation by hand, like loop unrolling specific register usage, etc. On the other hand, the interpreted code has been carefully tuned, because this tuning is giving us an insight of what we need for the specific task, and can therefore be regarded as one of the main tools that let to our results.

5 Conclusion

The performance comparison showed good results for the interpreter prototype. It is shown that in order to achieve good performance every possible optimization strategy has to be applied.

The next step is to reduce scheduling overhead by compiling the parts of the loop into machine native format during text parsing. This step is not very complicated and should yield to an significant speedup. This gives hope that the system
will stay in a small and manageable state, so it can easily be incorporated as already existing system as scripting extension language for DSP.

One drawback of the system is its reverse notation and other syntactically inherent problems, which make it hard to learn the language. Future work includes deciding on a final syntax. The goal is to keep the development system small by not introducing a complicated bytecode compiler, and therefore stay rather close to standard forth notation.

Further improvement in syntax and speed should be possible through higher level built-in vector functions and dedicated signal processing bytecodes.

A term goal would be to make the system independent from any other higher level computer music language.

References


