Software library for audio algorithm profiling

Istvan I. Papp, Vladimir R. Djurkovic, Zoran S. Marceta, Marko B. Janev and Dragan D. Kukolj, 
Member, IEEE

Abstract — This paper describes a software library intended to be used for tracking audio algorithm optimization in terms of resource requirements. The developed library makes possible creating a resource requirement snapshot at given development stage, and allows dimensioning the target platform capabilities according to the existing algorithm requirements. This approach has been successfully applied during optimization of complex speech enhancement algorithm for hands-free communication.

Keywords — Software, library, audio algorithm, profile, resources, memory, processing power, floating point, DSP.

I. INTRODUCTION

Wide range of modern consumer electronic devices includes digital signal processors (DSPs) usually performing multimedia and communication related tasks.

In order to be competitive and successful on the market, the devices have to use cost-effective solutions. This requires price-performance optimization as well in all details as in field of DSP.

Modern low-cost DSPs use most frequently fixed-point arithmetic, and they have on-chip memory. The implemented DSP algorithms have to be very well optimized in order to keep low memory footprint and required processor power, while offering satisfactory quality.

The development process of audio processing algorithms may be separated in two stages:

- Design - non-constrained step
- Optimization and implementation - constrained step

The non-constrained step involves algorithm design using tools like Matlab, Mathematica, C, etc. In this step, the algorithm developers are usually not faced with resource and processing power limitation (target platform restriction), although some common-sense constrains have to be respected. The goal is to develop an algorithm whose quality fulfills the requirements. At this stage, in order to speed up development, usually floating point arithmetic is used.

In the constrained step, the existing algorithm implementation has to be adapted in order to fit into target platform limitations. This may be done in two steps:

- In the first step, the processing power and utilized resources has to be reduced to level available at target platform. This step may introduce quality drop, but it must remain acceptable for the desired application.
- In the second step, the arithmetic has to be adapted to the abilities of the target platform (floating point or fixed point). This step shouldn’t influence quality significantly.

Figure 1. Algorithm development lifecycle

The audio algorithm development process described in this paper is shown in Figure 1. As it can be seen, two types of decision have to be made based on quality and required target platform resources, each requiring exact measures.
The quality of the output (processed) signal may be measured by subjective (e.g. listening test by human subjects) and objective measures (e.g. PESQ, SNR, ERLE) \cite{1}\cite{2}. Based on the measured results, decision is made whether the quality reaches the expected (required) level or not. As subjective test is more complex and tedious to perform, objective measure may be used to constantly monitor the quality during development process, without intensive subjective testing. Overall subjective checks may be carried out at main development milestones.

The estimation of the required resources at target platform may be a complex task, especially for complex algorithms. Yet, the estimation has to be reliably enough to make decision based on it. Underestimation of the resources causes pre-mature target platform implementation, thus involving unwanted return to design/optimization. Overestimation will consume unnecessary effort for optimization, thus lengthens the development time.

This paper describes a software component that measures the required processing power and memory by sampling, and collects information being used during arithmetic conversion to fixed point.

This approach was successfully applied in development of an integral algorithm for hands-free communication, consisting of echo cancellation, speaker localization, beam-forming, noise suppression and automated level control.

Section II describes the concept of the library. Section III deals with the implementation details, while the results are shown in section IV, followed by a conclusion in section V.

II. PROFILING LIBRARY CONCEPT

The estimation of required processing power and resources may be done in two basic ways:

- Algorithm analysis
- Algorithm execution sampling

Both approaches have their advantages and weaknesses.

Algorithm analysis has to be performed with great attention. The mistake probability rises with the complexity. By analysis, it is possible to reliably estimate the static behavior of algorithm, while the dynamic behavior can be sometimes only roughly estimated (e.g. in case of run-time changing parameters, decisions, etc.).

The advantage of algorithm analysis is in the discovery of theoretical worst cases. For range estimation, this approach may yield with arithmetic worst case, which is not necessarily the correct answer. For example, the analysis based on equation for a simple FIR filter of 1024 taps,

\[
y(n) = \sum_{i=0}^{1024} c(i) * x(n-i)\]  \hspace{1cm} (1)

may yield with range [-1024, 1024], however, it is known that the filter is designed to have output in range [-1, 1].

On the other hand, the execution sampling targets to describe the dynamic behavior. The results are run-time values for specific input.

The weakness of this approach is that the sampling has to be performed on a representative set of inputs, which covers all possible use cases. This is usually hardly achievable for complex algorithms.

By sampling, run-time characteristic may be determined, which not necessarily covers all possible inputs and worst cases, thus the most reliable approach is the combination of them.

For the example above (see Equation 1), the library may report range of [-0.5, 0.5]. During conversion to fixed point the range is extended to next higher power of 2 [-1, 1], which is the correct range.

This paper describes the sampling mechanism that aims to give as many information as possible, with the highest accuracy. It is intended to be used for programs written in C/C++, and which are using float arithmetic. It can be easily adapted to fixed point or integer arithmetic.

The idea is to replace the built-in type float and its arithmetic operations, and to collect operation statistic during execution. The collected information is:

- Number of float variable instances, which can be later interpreted as required memory.
- Number of basic floating point operations (addition, subtraction, multiplication and division)
- Number of calls to mathematical functions like sinus, cosine, logarithm, etc.
- Variable ranges.

After inclusion of (C++) library, the code can be compiled only with C++ compiler. The processing results must remain as were using built-in floating point type; differences are indication of a problem in the program code.

In order to cover all (or at least, most of) possible use cases, a set of test inputs has to be processed. The size of the set depends on algorithm complexity and features.

Every sampling session (processing of an input file) results with a report about the session. The report contains the collected data described above.

After running the set of inputs, all the reports are merged to summary report that contains the most accurate estimation of required resources and ranges (Figure 2).

The report shows how required resources change (increase or decrease). The information about maximal number of floats can be easily interpreted: as the mechanism takes into account even the local, temporary variables, the reported amount of memory shall be enough.
The programmer has to take care about input/output buffers, which could, but need not to be included into this number.

Regarding the executed operations, the decision is harder. Beside arithmetic floating point operations, there are other operations present in the algorithm, at least value transfers (writes, assignments). Assignments are not included into report because it is hard to distinguish between real transfer and temporary transfer introduced by the C++ compiler. Since on most DSP platform implementations such transfers do not occur, counting of them would significantly increase the overall transfers number, making the result incorrect. For example, for a FIR filter of 1024 taps there would be 1025 writes counted, while on DSP there would be only 1, while the other operations would refer to internal memory [3].

Instead, an experimental measure was used for scaling the report to real processing power. The targeted algorithm runs in real time on Pentium IV at 2.8 GHz, and performs 225 million of floating point operations per second (MFLOPS). As there is linear dependence between processor clock and power, the following equation may be set up:

\[ \text{PCIndex} = \frac{2800 \text{MHz}}{225 \text{MFLOPS}} = 12.5 \frac{\text{PCMHz}}{\text{MFLOPS}} \]

Taking into account the results shown in [4] that the efficiency of PC architecture and DSP is roughly about 1:3 when looking at the operating frequency, the final figure for DSP is as follows:

\[ \text{DSPIndex} = 12.5 \frac{\text{PCMHz}}{\text{MFLOPS}} \times 0.33 \approx 4.2 \frac{\text{DSPMHz}}{\text{MFLOPS}} \]

This shows that the initial algorithm has to be optimized almost 5 times (225 MFLOPS → 48 MFLOPS) to be able to run in real time on 200 MHz DSP (providing 48 MFLOPS), which results in reduction of floating point operations by same factor.

Once the amount of required resources fulfills the optimization requirements, the conversion to fixed point arithmetic may start. The conversion process starts with the introduction of data types that allows target platform arithmetic emulation on development platform. The next step is the definition of fixed point format for variables. At this point, the information about variable ranges becomes important, as directly defines the number of bits required for integer part within data word. After that, the code is converted to fixed point, and verified [5].

III. IMPLEMENTATION

The developed library relies on the following features of C++ language:
- classes
- operator redefinition
- function overloading
- macros

The library defines CFloat, a class that replaces the built-in C/C++ floating point data type. Class CFloat redefines all arithmetic operations appropriate to built-in floating point type, and also defines the appropriate conversions to and from other data types. This has to be made with precaution, because allowance of too many possible type conversions confuses the newest, stricter C++ compliant compilers. The type-replacement may be solved by a preprocessor directive.

The redefined operations are called every time a floating point operation should occur. Instead, the redefined operation is executed, and within it, beside the execution of actual floating point operation, the internal statistic is updated.

Also, the constructors and destructors are defined in manner that an internal counter always reflects the number of the existing CFloat instances, while another counter shows the maximal value reached in that session.

In order to provide complete information, all used library mathematical functions (included from math.h for example) have to be replaced with appropriate one, accepting CFloat as input argument instead of float or double.

Library mathematical function efficiency differs from implementation to implementation. Which way can be a call to mathematical function modeled by the adequate number of floating point operations, in order to calculate overall floating point operation number? If the manufacturer of the targeted platform provides such library, some information could be found in library documentation. In other cases (like there is no such library available or the target platform is still unknown), the number of analogous floating point operations has to be estimated.

In this project, the following approach was taken: the open source floating point library from GNU [6] was profiled using this library, and the results were included as analogous operations (see Table 1).

<table>
<thead>
<tr>
<th>Math function</th>
<th>Flops</th>
</tr>
</thead>
<tbody>
<tr>
<td>pow</td>
<td>67</td>
</tr>
<tr>
<td>atan2</td>
<td>295</td>
</tr>
<tr>
<td>cos</td>
<td>30</td>
</tr>
<tr>
<td>sin</td>
<td>30</td>
</tr>
<tr>
<td>sqrt</td>
<td>28</td>
</tr>
</tbody>
</table>

As the experience showed, memory copy and set functions are often used for copying data blocks. In order to ensure correct algorithm execution without code modification, these functions (memcpy, memmove and memset) are redefined for CFloat variables. Otherwise, usage of raw memory copy routines would destroy instance-specific statistics.

After algorithm execution, the statistics may be queried, printed or stored for later processing or merging. If the duration of the input sound is known, the number of executed operations may be averaged per second.

IV. RESULTS

This approach was used for optimization of algorithms intended to speech enhancement in hands-free
communications. The algorithm consists of the following blocks:

- Acoustic echo canceller (AEC)
- Audio localization (AL)
- Audio beam-forming (ABF)
- Noise suppression (NS)
- Automated gain control (AGC).

The target is to reduce memory footprint to 120k floating point numbers, and speed up the processing approx. 5 times, with no significant quality drop.

The optimization is done in several steps. Since AEC was the most demanding block, both in terms of memory and processing power, it was first optimized. Then, AL and ABF were improved, requiring significantly less processing power and some memory. Finally, the NS was replaced by an advanced algorithm that includes post-processing (PP), while the AGC was reintegrated into the optimized version.

The following figures show the profiling results during the optimization phases.

As it can be seen in Figure 3, the optimization of AEC module contributed at most, but as AEC was the block using most memory, this was expected behavior.

The processing power dropped in similar manner, approaching the targeted level after few optimization stages.

The optimization should not reduce the MOS quality of the output signal significantly. The speech quality was measured with PESQ MOS [1]. In Figure 5. MOS quality is shown the quality change during optimization phases. At the end of the optimization, the quality dropped from initial 2.6 to 2.5, which is acceptable.

![Figure 3. Memory requirements](image1)

![Figure 4. Required processing power](image2)

![Figure 5. MOS quality](image3)

V. CONCLUSION

On one hand, the developed library makes possible precise tracking of algorithm optimization in terms of required resources (memory and processing power). On the other hand, by adopting some assumptions, it can more-or-less precisely estimate of the required processing power and memory on the target platform. At the same time, it provides information that makes algorithm conversion to fixed-point arithmetic easier.

A disadvantage of the library is in slowing down the algorithm execution significantly as it inserts additional processing at every floating-point operation. This isn’t necessarily a flaw since the library does not collect real-time information, and it is used only in off-line operation mode, for making snapshots at current development stage. Of course, as the number of executed test vectors raises, the time overhead caused by library usage becomes longer.

The library makes possible to actually measure the progress of algorithm optimization for a complex speech processing system. As the final platform a DSP is targeted, the optimization had clear goals: memory footprint up to 120k floating point variables and processing power provided by a DSP at 200MHz, while retaining acceptable quality. According to the results, the optimization goals are achieved.

Further improvements may involve automatic insertion of required macros, replacement of the built-in mathematic library with open source library and possibly collection of additional information of interest.

LITERATURE