Enhance Processor Performance in Open-Source Applications

By David Katz [david.katz@analog.com]
Tomasz Lukasiak [tomasz.lukasiak@analog.com]
Rick Gentile [richard.gentile@analog.com]

As "open source" C/C++ algorithms become an increasingly popular alternative to royalty-based code in embedded processing applications, they bring new technical challenges. Foremost among these is how to optimize the acquired code to work well on the chosen processor. This issue is paramount because a compiler written for a given processor family will exploit that processor's strengths at the possible expense of inefficiencies in other areas. Performance can be degraded when the same algorithm is run directly out-of-the-box on a different platform. This article will explore the porting of such open-source algorithms to Analog Devices Blackfin® processors,[1] outlining in the process a "plan of attack" leading to code optimization.

What is Open Source?

The generally understood definition of "open source" refers to any project with source code that is made available to other programmers. Open-source software typically is developed collaboratively within a community of software programmers and distributed freely. The Linux[2] operating system, for example, was developed this way. If all goes well, the resulting effort provides a continuously evolving, robust application that is well-tested because so many different applications take advantage of the code. Programmers are encouraged to use the code because they do not have to pay for it or develop it themselves, thus accelerating their project schedule. Their successful use of the code provides further test information.

The certification stamp of "Open Source" is owned by the Open Source Initiative (OSI). Code that is developed to be freely shared and evolved can use the Open Source trademark if the distribution terms conform to the OSI's Open-Source Definition.[3] This requires that the software be redistributed to others under certain guidelines. For example, under the General Public License (GPL), source code must be made available so that other developers will be able to improve or evolve it.

What is Ogg?

There is an entire community of developers who devote their time to the cause of creating open standards and applications for digital media. One such group is the Xiph.Org Foundation,[4] a non-profit corporation whose purpose is to support and develop free, open protocols and software to serve the public-, developer-, and business markets. This umbrella organization (see Figure 1) oversees the administration of such technologies as *video*- (Theora), *music*- (the lossy Vorbis and lossless Flac), and *speech* (Speex) *codecs*.

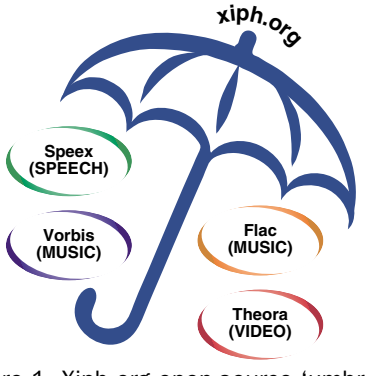

Figure 1. Xiph.org open-source 'umbrella'

The term *Ogg* denotes the container format that holds multimedia data. It generally serves as a prefix to the specific codec that generates the data. Vorbis, an audio codec we'll discuss here, uses Ogg to store its bitstreams as files, so it is usually called "Ogg Vorbis."[5] In fact, some portable media players are advertised as supporting OGG files, where the "Vorbis" part is implicit. Speex, a speech codec discussed below, also uses the Ogg format to store its bitstreams as files on a computer. However, *Voice over Internet Protocol* (VoIP) and other real-time communications systems do not require file storage capability, and a network layer like the Real-Time Transfer Protocol[6] (RTP) is used to encapsulate these streams. As a result, even Vorbis can lose its Ogg shell when it is transported across a network via a multicast distribution server.

What is Vorbis?

Vorbis is a fully open, patent-free, royalty-free audio compression format. In many respects, it is very similar in function to the ubiquitous MPEG-1/2[7] layer 3 (MP3) format and the newer MPEG-4[8] (AAC) formats. This codec was designed for mid- to high-quality (8-kHz to 48-kHz bandwidth, >16-bit, polyphonic) audio at variable bit rates from 16 to 128 kbps/channel, so it is an ideal format for music.

The original Vorbis implementation was developed using floating-point arithmetic, mainly because of programming ease that led to faster release. Since most battery-powered embedded systems (like portable MP3 players) utilize less expensive, more battery-efficient fixed-point processors, the open-source community of developers created a fixed-point implementation of the Vorbis decoder. Dubbed *Tremor*, the source code to this fixed-point Vorbis decoder was released under a license that allows it to be incorporated into open-source and commercial systems.

Before choosing a specific fixed-point architecture for porting the Vorbis decoder, it is important to analyze the types of processing involved in recovering audio from a compressed bitstream. A generalized processor flow for the Vorbis decode process (and other similar algorithms) is shown in Figure 2. Like many other decode algorithms, there are two main stages: front-end and back-end.

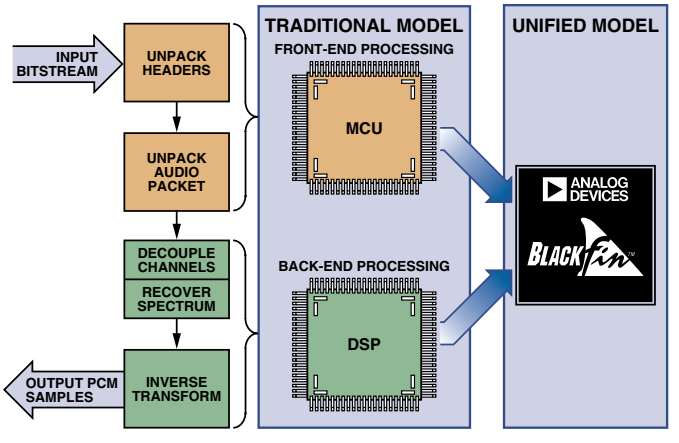

Figure 2. Generalized processor flow for the Vorbis decode process.

During the *front-end* stage, the main activities are header and packet unpacking, table lookups, and Huffman decoding. Operations of this kind involve a lot of conditional code and a relatively large amount of program space, so embedded developers commonly use microcontrollers for the front end.

Back-end processing is defined by filtering functions, inverse transforms, and general vector operations. In contrast to the front-end phase, the back-end stage involves more loop constructs and

memory accesses, often using smaller amounts of code. For these reasons, back-end processing in embedded systems has historically been dominated by full-fledged DSPs.

The Blackfin processor architecture unifies microcontroller (MCU) and DSP functionality, so there is no longer a need for two separate devices. It can be used efficiently to implement both front-end and back-end processing on a single chip.

What is Speex?

Speex is an open-source, patent-free audio compression format designed for speech. While Vorbis is used to compress all types of music and audio, Speex targets speech only. For that reason, Speex can achieve much better results than Vorbis on speech at the same quality level.

Just as Vorbis competes with royalty-based algorithms like MP3 and AAC, Speex shares space in the speech codec market with GSM-EFR and the G.72x algorithms, such as G.729 and G.722. Speex also has many features that are not present in most other codecs. These include variable bit rate (VBR), integration of multiple sampling rates in the same bitstream (8 kHz, 16 kHz, and 32 kHz), and stereo encoding support. Also, the original design goal for Speex was to facilitate incorporation into Internet applications, so it is a very capable component of VoIP phone systems.

Besides its unique technical features, Speex has the major advantages that it costs "nothing"—and can be distributed and modified to conform to a specific application. The source code is distributed under a license similar to that of Vorbis. Because the maintainers of the project realized the importance of embedding Speex into small fixed-point processors, a fixed-point implementation has been incorporated into the main code branch.

Optimizing Vorbis and Speex on Blackfin Processors

Immediate "out-of-the-box" code performance is a paramount consideration when an existing application, such as Vorbis or Speex, is ported to a new processor. However, software engineers can reap a big payback by familiarizing themselves with the many techniques available for optimizing overall performance. Some require only minimal extra effort.

The first step in porting any piece of software to an embedded processor like Blackfin is to customize the low-level I/O routines to fit the system needs. For example, the reference code for both Vorbis and Speex assumes that data originates from a file *and* processed output is stored into a file (mainly because both implementations were first developed to run on Unix/Linux systems where file I/O routines were available). In an embedded media system, however, the input and/or output are often connected to A/D and D/A *data converters* that translate between the digital and real-world analog domains. Figure 3 shows a conceptual overview of a possible Vorbis-based media player implementation. The input bitstream is transferred from a flash memory and the decoder output drives an audio DAC. Also, while some media applications (for example, portable music players) still use files to store data, many systems replace storage with a network connection.

When optimizing a system like the Vorbis decoder to run efficiently, it is a good idea to have an organized plan of attack. One possibility is to first optimize the algorithm from within C, then to streamline system data flows, and finally to tweak individual pieces of the code at an assembly level. Figure 4 illustrates a representative reduction of processor load through successive optimization steps and shows how efficient this method can be.

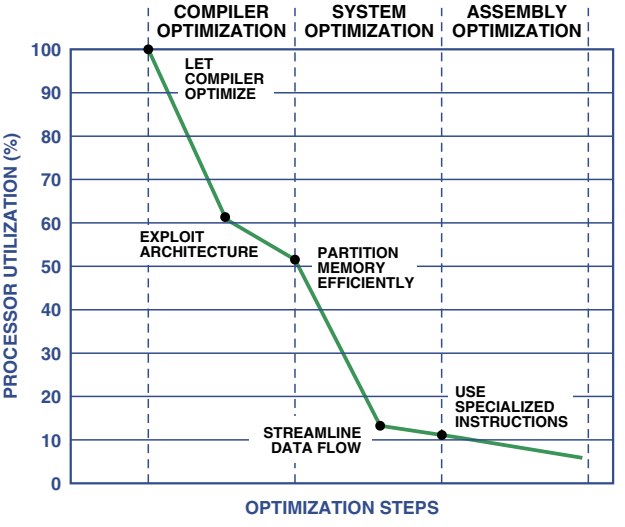

Figure 4. Steps in optimizing Vorbis source code on Blackfin, leading to significantly reduced processor utilization.

Compiler Optimization

Probably the most useful tool for code optimization is a good profiler. Using the *statistical* profiler in VisualDSP++ for Blackfin[9] allows a programmer to quickly focus on hotspots that become apparent as the processor is executing code. In many implementations, 20% of the code takes 80% of the processing time. Focusing on these critical sections yields the highest marginal returns. It turns out that *loops* are prime candidates for optimization in media algorithms like Vorbis because intensive number-crunching usually occurs inside of them.

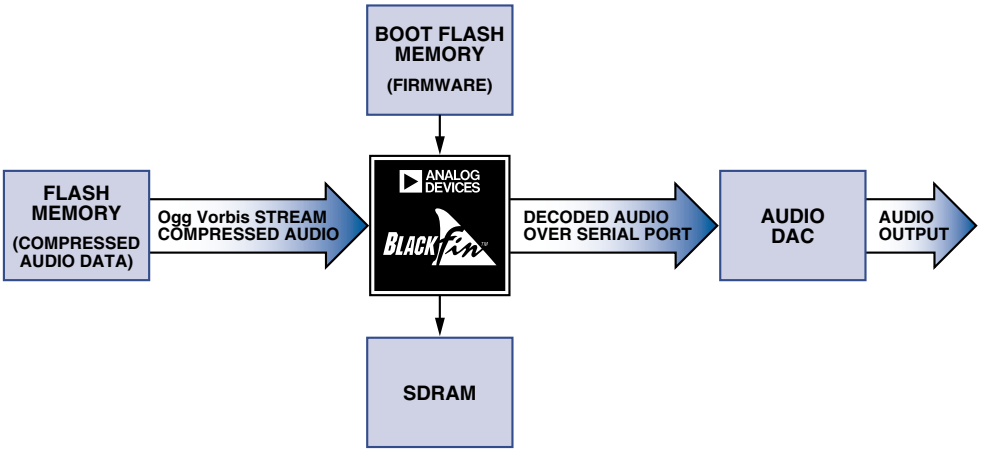

Figure 3. Example: Vorbis media player implementation.

There are also global approaches to code optimization. First, a compiler can optimize for either memory conservation or speed. Also, functions can be considered for automatic *inlining* of assembly instructions into the C code. (The compiler's `inline` keyword is used to indicate that functions should have code generated inline at the point of call. Doing this avoids various costs such as program flow latencies, function entry and exit instructions, and parameter passing overhead.) This, too, creates a tradeoff between space and speed. Lastly, compilers like the one available for Blackfin can use a two-phase process to derive relationships between various source files within a single project to further speed up code execution (*inter-procedural analysis*).

As mentioned above, most reference software for media algorithms uses floating-point arithmetic. But software written with fractional fixed-point machines in mind still misses a critical piece. The language of choice for the majority of codec algorithms is C, but the C language doesn't "natively" support the use of fractional fixed-point data. For this reason, many fractional fixed-point algorithms are *emulated* with integer math. This may make the code highly portable, but it doesn't approach the performance attainable by rewriting some math functions with machine-specific compiler constructs for highest computational efficiency.

A specific example illustrating this point is shown in Figure 5. The left column shows C code and Blackfin compiler output for emulated fractional arithmetic that works on all integer machines. One call to perform a 32-bit fractional multiplication takes 80 cycles. The right column shows the improvement in performance obtainable by utilizing (`mult_fr1x32x32`), an intrinsic function of the Blackfin compiler that takes advantage of the underlying fractional hardware. With this fairly easy modification, an 86% speedup is achieved.

System Optimization

System optimization starts with *proper memory layout*. In the best case, all code and data would fit inside the processor's L1 memory. Unfortunately, this is not always possible, especially when large C-based applications are implemented within a networked application.

The real dilemma is that processors are optimized to move data independently of the core via *direct memory access* (DMA), but MCU programmers typically run using a cache model instead. While core fetches are an inescapable reality, using DMA or cache for large transfers is mandatory to preserve performance.

To introduce the discussion, let's consider several attributes inherently supported by the Blackfin bus architecture. The first is the *ability to arbitrate requests without core intervention*. Because internal memory is typically constructed in sub-banks, simultaneous access by the DMA controller and the core can be accomplished in a single cycle by placing data in separate banks. For example, the core can be operating on data in one sub-bank while the DMA is filling a new buffer in a second sub-bank. Under certain conditions, simultaneous access to the same sub-bank is also possible.

There is usually only one physical bus available for access to external memory. As a result, the arbitration function becomes more critical. Here's an example that clarifies the challenge: on any given cycle, an external memory location may be accessed to fill the instruction cache at the same time that it serves as the source and destination for incoming and outgoing data.

Instruction Execution

Blackfin processors use hierarchical memory architectures that strive to balance several levels of memory having differing sizes and performance levels. On-chip *L1 memory*, which is closest to the core processor, operates at the full clock rate. This memory can be configured as SRAM and/or cache. Applications that require the most determinism can access on-chip SRAM in a single core clock cycle. For systems that require larger code sizes, additional on-chip and off-chip memory is available—with increased latency.

SDRAM is slower than L1 SRAM, but it's necessary for storing large programs and for data buffers. However, there are several ways for programmers to take advantage of the fast L1 memory. If the target application fits directly into L1 memory, no special action is required other than for the programmer to map the application code directly to this memory space—as in the Vorbis example described above.

If the application code is too large for internal memory, as is the case when adding, say, a networking component to a Vorbis codec, a caching mechanism can be used to allow programmers to access larger, less expensive external memories. The *cache* serves as a way to automatically bring code into L1 memory as it is needed. Once in L1, the code can be executed in a single core cycle, just as if it were stored on-chip in the first place. The key advantage of this process is that the programmer does not have to manage the movement of code into and out of the cache.

The use of cache is best when the code being executed is somewhat linear in nature. The instruction cache really performs two roles. First, it helps pre-fetch instructions from external memory in a more efficient manner. Also, since caches usually operate with some type of "least recently used" algorithm, instructions that run the most often are typically retained in cache. Therefore, if the code has been fetched once and hasn't yet been replaced, it will be ready for execution the next time through the loop.

ORIGINAL

```
int32 MULT31(int32 a, int32 b) {
 int32 c;
 c = ((long long)a * b) << 1;
 return c;
}

R2 = R0 ; // lo(a)
R0 = R1 ; // lo(b)
R1 >>>= 0x1f ; // hi(a)
R3 = R2 >>> 31 ; // hi(b)
[ SP + 0xc ] = R3 ;
CALL ___mulli3 ; // 64x64 mult (43 cycles)
R2 = 1 ;
R3 = R2 >>> 31 ;
[ SP + 0xc ] = R3 ;
CALL ___lshftli ; // 64 shift (28 cycles)
P0 = [ FP + 0x4 ] ;

80 cycles
```

IMPROVED

```
fract32 MULT31 (fract32 a, fract32 b) {
 fract32 c;
 c = mult_fr1x32x32(a, b);
 return c;
}

A1 = R0.L * R1.L ( FU ) || P0 = [ FP + 0x4 ] ;
A1 = A1 >> 16 ; // use of accumulator
R2 = PACK ( R0.L , R1.L ) ;
CC = R2 ;
A1 += R0.H * R1.L ( M ) , A0 = R0.H * R1.H ;
CC &= AV0 ;
R2 = CC ;
A1 += R1.H * R0.L ( M ) ;
A1 = A1 >>> 15 ;
R0 = ( A0 += A1 ) ;
R0 = R0 + R2 ;

11 cycles (14%)
```

Figure 5. Compiler intrinsic functions are an important optimization tool.

Wary real-time programmers have not trusted cache to obtain the best system performance because system performance will be degraded if a block of instructions is not in cache when needed for execution. This issue can be avoided by taking advantage of *cache-locking* mechanisms. When critical instructions are loaded into cache, the cache lines can be locked to keep the instructions from being replaced. This allows programmers to keep what they need in cache and allow less-critical instructions to be managed by the caching mechanism itself. This capability sets the Blackfin processor apart from other signal processors.

Data Management

Having discussed how code is best managed to improve performance on this application, let's now consider the options for data movement. As an alternative to cache, data can be moved in and out of L1 memory using a DMA controller that is independent of the core. While the core is operating on one section of memory, the DMA is bringing in the next data buffer to be processed.

The Blackfin data-memory architecture is just as important to the overall system performance as the instruction-clock speed. Because there are often multiple data transfers taking place at any one time in a multimedia application, the bus structure must support both core and DMA accesses to all areas of internal and external memory. It is critical that arbitration of the DMA controller and the core be handled automatically, or performance will be greatly reduced. Core-to-DMA interaction should only be required to set up the DMA controller, and later to respond to interrupts when data is ready to be processed. In addition, a data cache can also improve overall performance.

In the default mode, a Blackfin performs data fetches as a basic core function. While this is typically the least efficient mechanism for transferring data, it leads to the simplest programming model. A fast scratchpad memory is usually available as part of L1 memory; but for larger, off-chip buffers, the access time will suffer if the core must fetch everything. Not only will it take multiple cycles to fetch the data, but the core will also be busy doing the fetches.

So, wherever possible, DMA should always be employed for moving data. Blackfin processors have DMA capabilities to transfer data between peripherals and memory, as well as between different memory segments. For example, our Vorbis implementation uses DMA to transfer audio buffers to the audio D/A converter.

For this audio application, a "revolving-door" double-buffer scheme is used to accommodate the DMA engine. As one half of the circular double buffer is emptied by the serial port DMA, the other half is filled with decoded audio data. To throttle the rate at which the compressed data is decoded, the DMA interrupt service routine (ISR) modifies a semaphore that the decoder can read—in order to make sure that it is safe to write to a specific half of the double buffer. In a design that lacks an operating system (OS), polling a semaphore means wasted CPU cycles; however, under an OS, the scheduler can switch to another task (like a user interface) to keep the processor busy with real work.

The use of DMA can lead to incorrect results if data coherency is not considered. For this reason, the audio buffer associated with the audio DAC is placed in a noncacheable memory space, since the cache might otherwise hold a newer version of the data than the buffer to be transferred by the DMA.

Assembly Optimization

The final phase of optimization has to do with rewriting isolated segments of the open-source C code in assembly language. The best candidates for performance improvement by an assembly rewrite are usually interrupt service routines (ISRs) and reusable signal-processing modules.

The impetus for writing interrupt handlers in *assembly* is that an inefficient ISR will slow the responses of other interrupt handlers. For example, some audio designs must use the audio ISR to format AC97 data bound for the audio DAC. Because this happens periodically, a long audio ISR can slow down responses of other events. The best way to reduce the interrupt handler's cycle count is to rewrite it in assembly.

A good example of a *reusable signal-processing module* is the *modified discrete cosine transform* (MDCT) used in back-end Vorbis processing to transform a time-domain signal into a frequency domain representation. The compiler can never produce code as "tight" as a skilled assembly programmer can, so the C version of the MDCT is inefficient. An assembly version of the same function can exploit the hardware features of the Blackfin architecture, such as single-cycle *butterfly add/subtract and hardware bit-reversal*.

Today, Blackfin ports of both Vorbis and Speex exist and are available on request. These ports run on the ADSP-BF533 EZ-KIT Lite.[10] An open-source port of μClinux is also available at blackfin.uclinux.org.[11] Together, these enable a wide variety of applications that seek to integrate royalty-free speech or music capability while retaining plenty of processing room for additional features and functionality. For example, the new ADSP-BF536/ADSP-BF537,[12] with its integrated Ethernet MAC, opens the door to low-cost networked audio and voice applications. Clearly, open-source code heralds a revolution in the embedded processing world, and Blackfin processors are poised to take full advantage of this situation.

FOR FURTHER READING

Parris, Cliff and Phil Wright. *Implementation Issues and Tradeoffs for Digital Audio Decoders*. Monmouthshire, UK: ESPICO, Ltd.

REFERENCES—VALID AS OF FEBRUARY 2005

[1] http://www.analog.com/processors/processors/blackfin/
[2] http://www.linux.org/
[3] http://www.opensource.org/docs/definition.php
[4] http://www.xiph.org/
[5] http://www.vorbis.com/
[6] http://www.ietf.org/rfc/rfc1889.txt
[7] http://www.chiariglione.org/mpeg/
[8] http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm
[9] http://www.analog.com/en/prod/0,2877,VISUALDSPBF,00.html
[10] http://www.analog.com/en/prod/0,2877,BF533-HARDWARE,00.html
[11] http://blackfin.uclinux.org/
[12] http://www.analog.com/Analog_Root/static/promotions/blackfin750/index.html