Pygar: How to Process Music in Parallel

Laurel Pardue & Robert McIntyre

May 14, 2010

Abstract

Digital audio mixing is really hard for everyday musicians with normal systems for one reason: the individual voices that make up a performance want to be processed in parallel, but current systems handle them mostly in serial. To address these concerns we've created Pygar, a system on the FPGA to process audio like a software mixer, but using a dedicated processor for each voice. Each processor leads its own transformation program which can be written in a high level language like C. This easy parallelization works because voices don't interact with each other. Our system is extensible and even in its current implementation can handle 16 separate voices — something which normally requires expensive (~$10,000) hardware acceleration.

The Problem

Say that you want to record your band's music in a high quality digital format: One thing you could try is to put a single microphone in the center of the stage and hope for the best. This is bound to cause disappointment, because everyone ends up too far away from the recording device, reducing the quality of the recording. Also each player's volume will be determined by their distance from the microphone, while the ideal is to have each players volume normalized with respect to the overall performance. A much better alternative to the single-microphone approach is to give each player a microphone that is only a few inches from their instrument and then combine all the separate recordings (*voices*) into a single stream. Some instruments, such as drums, may even receive up to six microphones! Lots of microphones, each very close to the instrument they are recording, solves the problem of attaining good quality audio. But, each voice still needs to be processed a little so that they will all sound good together. The most common type of correction is just changing the volume so that every instrument sounds correct with regard to the total performance, but sometimes other corrections such as re-verb or slight timing alterations are also required. The process of fixing the voices to sound good together and then combining them all into a single track is called *mixing.*

Hardware mixers with hardware effects racks were the original solution to the challenges of audio and until recently, remained the common means for implementation. However, above more than 12 or 16 mono voices the price of a hardware mixer gets exceedingly high. For effects, the audio is sent off board to another large hardware processing unit. This requires lots of cabling and the quickly claims all the physical input output ports on the mixer. Software emulation of the hardware is quickly replacing hardware for unified processing and mixing. Software emulation is still challenged. Even with today's high speed computers, a handful of tracks with moderate numbers of effects will strain the average user's processor. Today mixing is done with software, but even the best software is limited in the number of voices it can handle without *clipping* (loss of audio data). Clipping is very noticeable and completely unacceptable for audio processing. Software solutions to the audio mixing problem are all limited by the serial nature of the computer, which in the end just jumps from voice to voice as fast as it can in an attempt to handle everything at once. The best software today on laptop systems can't handle more than about 10 voices at once. For portable options using an embedded environment, few low end micro-processors can achieve more than a few voices and more than very basic processing. This is a real problem, because bands and other groups like T.V. shows can easily top 25+ voices, or even 100 voices if synthetic digital instruments or sound effects are used. How do T.V. shows, movies, and huge bands do all this processing? The current state-of-the-art is extremely expensive hardware acceleration for the best software mixers. Such systems tend to *start* at around $10,000. We want to keep the good parts of software mixers, namely the ability to easily describe algorithms to manipulate voices in software, while respecting the nature of the problem, which is parallel operations on almost independent voices with real-time timing constraints.

To make a good replacement for a mixer, you first have to understand how software mixers work.

Anatomy of a Mixer

The easiest way to understand what a mixer does it to look at one. Figure 1 shows a reasonably simple mixer setup in ProTools. ProTools is generally considered the most powerful audio editing software available with one of the best mixers. It will help demonstrate how the mixer works and what it does. First, the screen shot includes three audio inputs, two sub-mixes, and the master mix (bottom line). The three audio inputs are two mono inputs, two stereo inputs (note two line level displays) where two mono audio streams are linked together, and two *instrument tracks* (to clarify track vs. stream, a mono track has one audio stream and a stereo track has two). Unlike normal audio inputs that are essentially straight data feeds, instrument tracks are a set of instructions that either trigger virtual instruments based on processed samples or control completely synthetic sound generation. Next consider the *I/O tab*. This consists of input options and output options. Generally input is either a pre-recorded audio stream or a live feed from hardware. In the case of an instrument track it is a sequence of instructions intended for the synthetic instrument. The output option directs where the audio output goes. Most tracks will go to the master *fader* (volume control) unless specifically directed elsewhere. Routing output

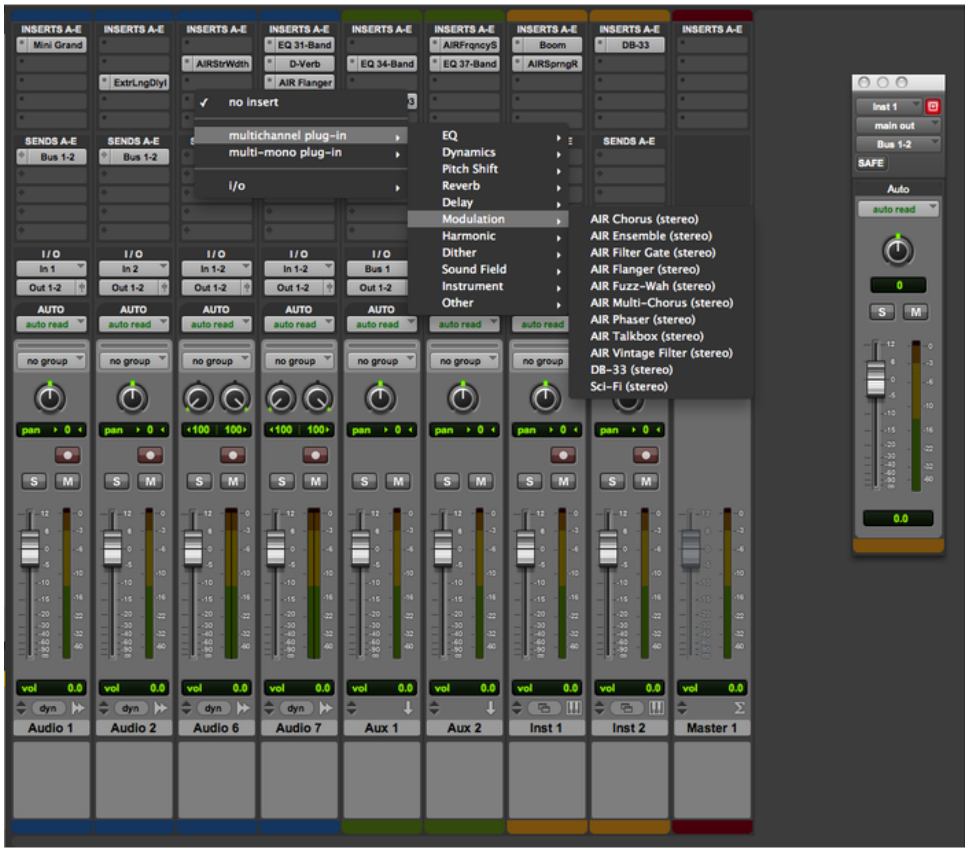


Figure 1: Screenshot of ProTools in action. ProTools is one of the best programs for mixing audio.

to an *aux track* instead of the master allows a track to be pre-mixed with additional tracks and effects before arriving at the master mix. For instance, if five microphones are recording a drum kit, the aux track can be used to set the drum volume using only one fader. Alternatively a track can be directed to a specific hardware port. One last routing option found above i/o is the *send*. The send is similar to main i/o in that it allows the audio to be passed somewhere like an aux track, except that the primary track output still exists. Essentially it allows routing one stream to many places, while i/o directs to only a single destination.

Above the sends are the *inserts*. These are any affect algorithms that should be run on the audio stream. Such algorithms could be delay, reverb, modulation, analysis tools, etc. These can often be exceedingly computationally intensive. *Virtual Instruments* are included as an insert as the sound generation, whether sample based or not, will require processing and access memory differently. Professional audio software systems typically run these Inserts on external DSPs using hardware assistance to route the substantial simultaneous memory requests and a hardware mixer to combine resulting audio streams. The example Insert menu shown demonstrates the difference ways of handling stereo. Stereo (and other multi-voice formats like Dolby) can be treated as multiple separate audio tracks (multi-mono) or as a single unit (multichannel). Multichannel algorithms like reverb usually will alter one audio stream's output based on another audio stream within the track. Lastly, all audio tracks will have *pan* (which basically sets relative volumes on the different sides of a stereo track, or mono track routed as a stereo track) and *volume*. The *master track* sums everything together, supports processing (like compression) on the final audio stream and sends it to the correct audio out.

Mixers in the Real World

A typical recording session might involve recording and or playing multiple voices simultaneously. A standard 4 piece band might take 6 microphones for the drum, two microphones for each guitar, one microphone for the singer and, two more microphones for bleed and ambiance. As you wouldn't want the same audio filters applied to the vocals as the bass guitar, each microphone voice is shaped individually for optimal sound on that voice before being added to the whole which is what you hear as the listener. Another example centered on playback might be instrument emulation. If you want to emulate the playing and sound of a harp electronically using a different interface, the system might play back a separate sample or synthesis for each of the possible 46 struck strings which is individually shaped based on the trigger style and timing and with all the notes then summed to generate the single instrument audio. Electronic production is usually done processing a number of pre-recorded voices and generative audio sources (sampler or synthesizers) and summing together again. The common aspect in all these, from live recording to pre-recorded playback is that each audio voice starts out as a separate stream requiring independent processing ideal for parallel computation. Audio is also a steady sample rate of at least 44.1kHz and if being performed live, must not suffer from substantial latency. Each voice involves a steady data flow. Therefore, audio playback must source large quantities from memory or, in the case of live recording, must digitize the multiple analog sources and must pass that data along. Taking the four piece band with 12 voices, using a minimal recording standard of 44.1kHz and a bit-depth (sample size) of 16 bits, this ends up being 8Mb/s. This may seem reasonable except that audio is fairly error intolerant. It is not considered acceptable to easily drop a sample or re-send it because a timing deadline was missed. That 8Mb/s service must be guaranteed. This 8Mb/s rate is for a simple 4 piece band recording at minimal data rates. Take the example of a movie with an orchestral sound track and a busy scene with lots of foley (sound effects). This might involve 100 voices at 96kHz and 24 bits per sample necessitating a consistent over 200Mbs data rate. Once the data is in the system, mathematical processing of an audio stream for effects or synthesis is commonly computationally intensive. Audio effects frequently involve Fourier Transforms and other serious math. Again, due to the timing deadlines, processing algorithms must complete within one sample clock. Don't forget that these additional audio processes will add further memory bandwidth requirements. The last probably most straight forward part of the audio system, is the mixer. At its most simple, the mixer takes each individual voice, balances it with the other voices by scalar multiplication and then sums all the voices together to achieve the whole. Normally there is actually some smart math done here. The mixer is not typically a straight addition, otherwise adding 16 voices and not overwhelming the output would require an unacceptably low input volume per voice. Lowering the volume effectively reduces bit-depth and loss of information. More complicated mixers allow routing of audio between and to different voices. For instance, the drums are composed of six individual voices which are convenient to sum to a new voice so that the beat has a single volume control or blended reverb. Grouping voices before the final mixer is really the closest the audio streams come to interacting with each other and still follows the concept of independent parallelizable voices unless side-channeling is allowed. This is where an audio file is used as an input control to an effect on another stream. An example of this is called gating. If the bass guitar is doing a continuous bass line, the intermittent bass drum might get lost or muddy. A good means to clean this up and retain the bass drum punch is to use the bass drum as a gating input on the bass guitar line so that each time the bass drum is hit, the guitar's volume is temporarily reduced and as it is not a two way dependence, does not negatively impact the parallel nature of the audio flow.

Our Ideal System

Now that we know the problems faced by mixer systems in the real world, it is clear what a replacement system should be able to do. Our system operates on two guiding principles: parallel audio processing and programmable audio processing. To this end our ideal architecture uses a string of configurable processors for each voice as follows:

The first component is a control module. This has knowledge of the overall routing scheme. It uses this knowledge to issues instructions on which streams to play and when (DMA), and passes controls onto other modules such as the processing units and mixer. Example controls might be volume scalars for the different voices sent to the mixer or the delay length in a processor running a delay algorithm. The second module is a memory fetch module that, given instructions from the control module, coordinates requests to main memory or accepts input from A/Ds. The memory module must be able to provide one sample per live voice every sample clock cycle in order to maintain sample rate integrity. The third modules are the processing cores. These take the sample streams and run a dynamically loadable algorithm on the sample stream. They also take instructions from the control module on any effect controls. In Figure 2, the soft-cores are essentially implementations of a plugin. Lastly, everything collects in the mixer module for scaling and summation.

When the processing starts, the DMA reads in each voice and sends the audio data ("samples") off to a chain of soft-cores. The DMA knows to which chain the samples should be forwarded by looking up the samples' voice in a hash table read at startup. Once the samples enter the soft-core chain, the first soft-core in line takes the samples and performs some sort of audio operation, producing modified audio samples. These samples feed into the next soft-core which performs a different algorithm. As condition for functional correctness, we require the algorithm that each soft-core contains to be able to produce audio samples at the desired 44khz rate, but we allow an arbitrary setup time for the soft-core to get ready. That way, a pipelined processor can take its time and fill up its pipeline completely with no worries. Once the soft-core starts producing samples, it must produce them at a rate of at least 44khz forever. If every soft-core in the chain obeys this timing condition, then the whole chain will as well, so the entire chain of soft-cores can be treated as a single big soft-core for timing purposes. Our soft-cores are just Lab 5 processors preloaded with specific audio processing algorithms.

Eventually, the samples make their way to the end of the chain of soft-cores. Every chain of soft-cores ends in a FIFO. Each of the FIFOs leads to the mixer, which only begins operation once all the FIFOs are full. Because of the timing constraints we impose on our soft-cores, once every FIFO has at least one sample, all of them will forever more be able to supply samples at 44khz. The mixer therefore waits until all the FIFOs have elements and then reads samples from all FIFOS at a rate of 44khz. The mixer combines the samples through simple addition and scaling, creating a single stream of sample data, which is sent back from the FPGA to the host computer where we play the stream and/or save it to a file. The mixer ensures that it outputs samples at a rate of 44khz by running everything through its own internal buffer FIFO clocked at 44khz.

Our Actual System

Due to time limitations, we went for a more simplified architecture than our proposed ideal system. First, we are not doing any dynamic control passing, so there is no need for a control module. Rather than a memory request system, we used existing tools to implement sample passing between the host and the FPGA which is host driven. This alleviated the need to setup a sample clock within our present design. This was a good short term design simplification that also made for easy coordination between the host system and FPGA since the host only sends when the FPGA is ready, which implicitly guarantees that no samples from memory are missed. We presently have opted for a one-deep parallel set of SMIPS soft-cores to provide processing capabilities (Figure 3). That is, each voice is still processed in parallel, but only passes through one processor. The mixer functionally is the same as the general architecture except that there is no synchronization to a sample clock. Instead, the FPGA sends the mixed signal back to the host for writing to a file. The implementation of our architecture was done within AWB (Architect's Workbench). This let us use the Intel ScratchPad

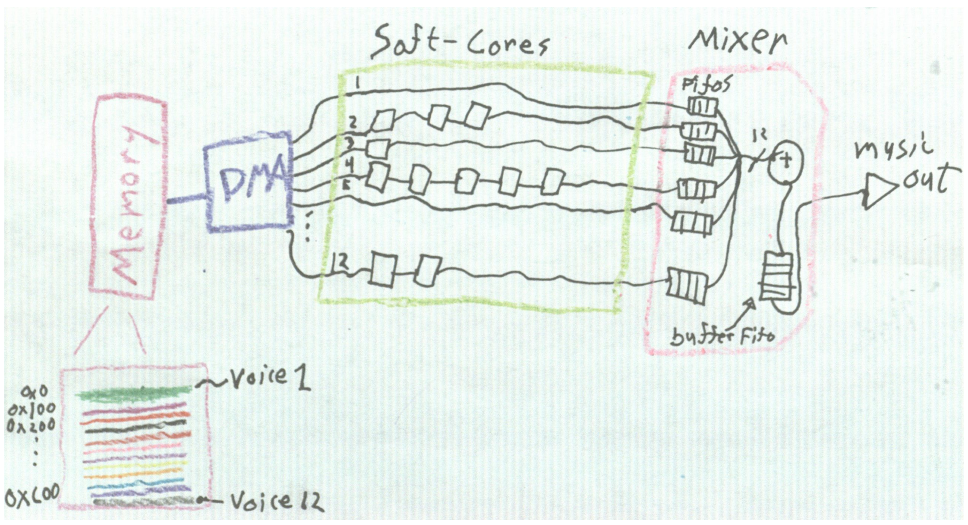


Figure 2: The audio data *samples* start in the memory, but are soon pulled into action by the DMA (direct memory access). The DMA sends the samples to a chain of 0 or more *soft-cores*, where they are transformed according to the soft-cores' algorithms. After running the gauntlet of soft-cores, the samples flow first to a buffering FIFO, and finally to a *mixer*, which sends the samples off to be played by speakers or stored in a file.

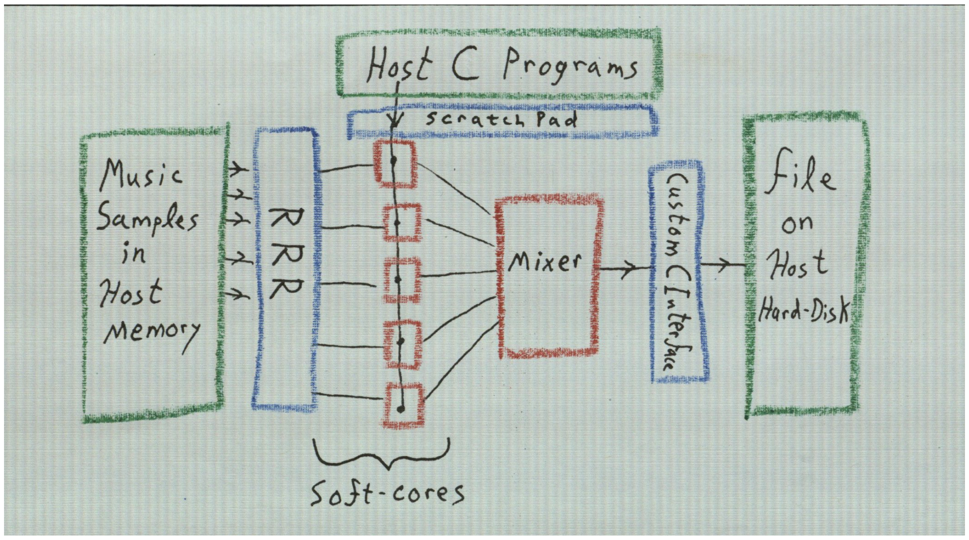


Figure 3: Our Current Implementation is a reduced version of our Ideal Implementation. Instead of a configurable grid of processors, we use a one-deep set of processors. Songs are taken strictly from host memory via RRR and are fed back to the host to be written to a file.

caching system and RRR. ScratchPad is for loading programs into the soft-cores, while RRR enables fast loading of music samples from host memory. RRR has a higher bandwidth than SCEMI which will be helpful for properly testing large multi-voiced systems and despite some initial hitches ScratchPad has proved to be very effective for servicing soft-core memory needs. Each soft-core gets its own ScratchPad, simplifying our memory interactions nicely.

A big part of our present architecture was code reuse. Early labs in class and in the 6.375 AWB package demonstrated how to stream audio from the host, through the host-FPGA interface, and route it to an FFT for processing. We are not presently using the FFT, but are reusing much modified streaming code in both C++ and bluespec. The soft-core is the soft-core implemented in lab 5 with modifications to use the RRR abstraction, an added interface for audio samples, and additional instruction set response to pass the samples to the SMIPS code on the core. In summary, our system was implemented with minimal bluespec and three main re-used components: RRR, ScratchPad, and Lab 5 SMIPS processors.

Bluespec Implementation

Bluespec implementation was split into code modules. This first is the primary audio pipeline module `audioCorePipeline.bsv`. It is responsible for creating the cores and creating the links between audio in, cores, and the mixer and supports passing debug information back to the host. Since the DMA portion of the generalized architecture was abandoned for a fairly direct pipe-in, this module not only handles connections, it also handles the remaining memory functions. The left over functions are largely the immediate interface protocols to get and retrieve samples through the RRR and correctly route them to the right core. One more interesting aspect of the main pipeline module is the synchronization of the audio samples. It is necessary to make sure that the sample from voice 1 sent at time T is the voice 1 sample added to the sample from voice 2 sent at time T. Software must resort to time stamping to make sure that sample order and synchronicity is achieved. A benefit of hardware is that as long as the samples start in order, they are guaranteed to flow in order and we can wait for the relevant mixer FIFOs to receive a sample before mixing. For files of different length, this approach adds a bit of complexity. If the hardware waits for the sample from each voice before firing and one of the voices is no longer sending, the FPGA will be hung. This is addressed by the input handling sending Invalids once an end of file is reached and the file stops sending. This required modifying the interfaces used by the audio stream to the cores and the mixer along with both understanding how to correctly continue the pass thru of the Invalids. For instance, the core shouldn't immediately start forwarding Invalids to the mixer when an EOF is reached. The core may still be processing valid samples and to keep order, Invalids should not forward until the core completes sample processing. A last nicety of the pipeline bluespec code structure is that it is well parametrized for scaling up to 26 cores. Changing one define from 1 to 16 and a recompile is all that is required to add 15 cores.

Mixer

The mixer is fairly simple. Since we are not implementing dynamic controls presently, it takes as an argument a set of mixer voice scalars which it uses to scale each voice appropriately. The scalars are set at number 0-255. The mixer multiplies the sample by the voice streams scalar and then returns it to the same absolute scale by dividing by 256. This is a simple efficient right shift. It is normal mixers scale down the volume of voices while combining them. A stream is assumed to be at full volume when raw. Scaling it above it's original size is considered more like running through a specific amplifier which is a plug-in, not a mixer function. The algorithm for summing voices is similar in nature. It uses a larger bit space to add the 16 bit samples together raw and then bit shifts the result back into the original 16 bit base. There are definitely better options than this. As previously mentioned, sample order is preserved by the axiom that each voice goes in order so once all voices have supplied a sample, the output is ready to start. The mixer also tracks voice completion through End of Files and returns the final process ending EOF.

ScratchPad

The most substantial code reuse in Pygar was Intel's ScratchPad. This system solves the problem of dynamically loading C programs into the memory of each

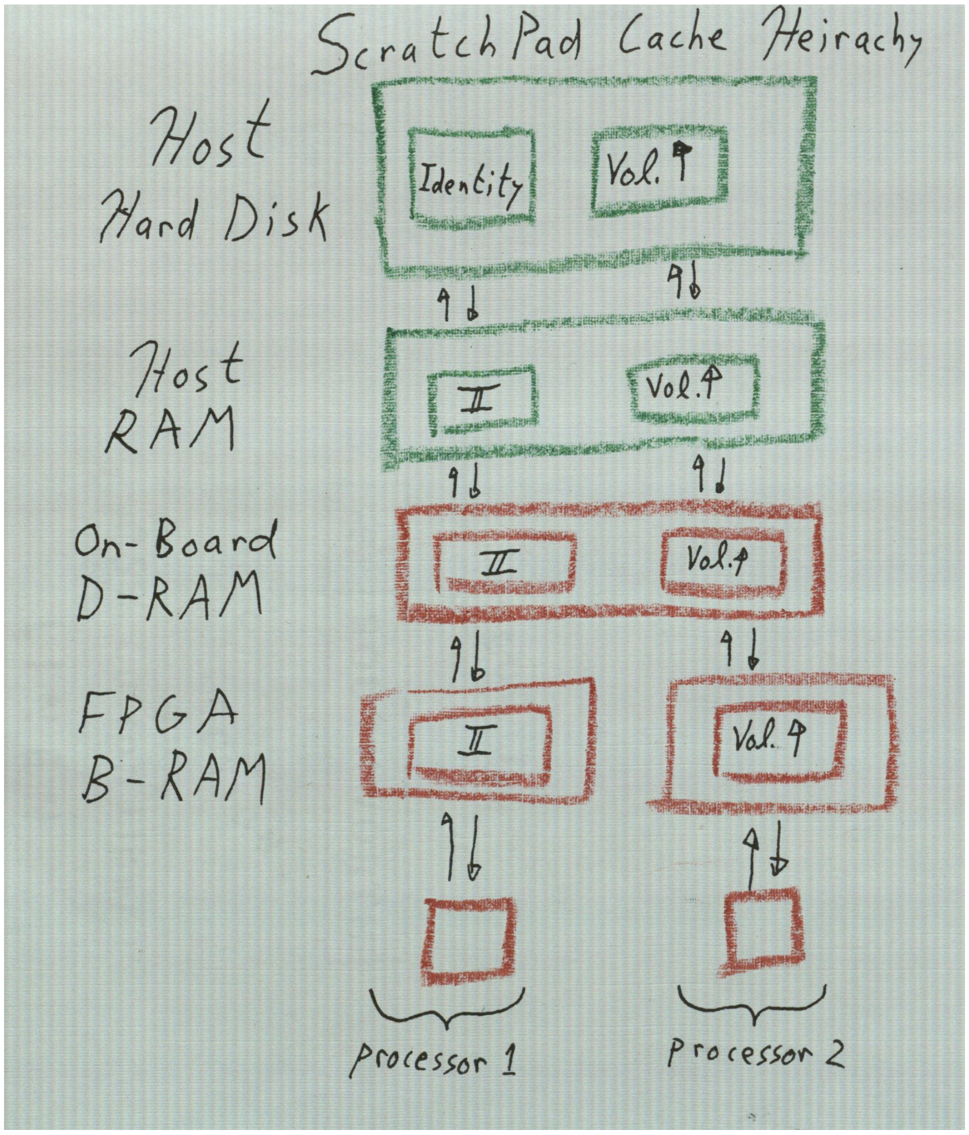


Figure 4: When a soft-core processor tries to read its program, it cache-misses all the way up to host memory. Subsequent memory requests are fast since the cache is initialized. Each processor operates in its own memory space which is completely independent from all other processors.

soft-core processor. A ScratchPad is a cache hierarchy which extends all the way from the FPGA soft-core to the hard-disk of the host machine. We easily select different C programs to be executed by the soft core by pointing the hard-disk level of the cache to our program. Then, when the Processor begins to access its local memory on BRAM, the cache misses all the way up to the hard-disk and reads the target C program. Subsequent memory accesses are much faster since the cache is initialized after the first miss. As long as the entire program fits into the FPGA's on-board DRAM, our timing constraints will be satisfied. Each processor receives its own ScratchPad, but each ScratchPad shares a common cache in DRAM and all higher levels. This gives us efficiently implemented separate memory spaces for each processor. To switch programs, we simply point the ScratchPad Cache hard-disk level base to a different location in memory, and the cache takes care of flushing old data automatically.

Transformation Programs

We have written a handful of very simple SMIPS programs as a proof of concept. They are thru or identity which simply returns the original sample, louder, softer, and bit-shifter. Louder acts as an amplifier such as mentioned in the mixer discussion. It is coded to mimic clipping as scaling a sample above the playable analog volume results in clipping rather than letting the number cycle over to something even more unhelpful. Softer is self explanatory. Bit shift does a bit reduction of the song from 16 bits to 8. Go Nintendo.

Lab 5 Processors

The core consists of two modules: Core (`audioCore.bsv`) and Processor (`Processor.bsv`). The Core wraps the Processor, and is virtually the same as the Lab 5 core except that it makes memory connections with the ScratchPad instead of through the host. Keeping the ScratchPad in the core also means that non-soft-cores can easily be substituted for a core at the pipeline generation stage. Otherwise, the core passes the audio samples on to its processor. The Processor (`Processor.bsv`) is derived from the Lab 5 processor as stated. The decision to use this particular processor was influenced by the fact that it was familiar code, it achieves the objective, and there are simple tools for successfully using it. Due to a bug in Xilinx's elaboration tools, we were forced to make several trivial modifications, but the only additional functionality we added to the processor was the ability to pass audio samples between the SMIPS program and the core. We achieved this by adding two new sub-instructions to the `MTC0` and `MFC0` op-codes. Seeing as the `MTC0` and `MFC0` are designed for this type of special register passing, this seemed like a far more logical solution than adding a new opcode. On the SMIPS-to-core side (`MFC0`), the first sub-instruction relates whether End of File has been hit (this automatically clears for effective state reset) while the second sub-instruction passes the valid sample. Right now, these requests are not serviced if no valid data (24 bit samples) enter the processor. Bad samples only occur from bad behavior by the ScratchPad C program, so while the current stalling is reasonable, it might be better still to add an error flag in this situation. The two `MTC0` instructions exist to store a sample manipulated by C, and to flag the completion of the C program, signaling the processor to signal End Of File. The 'complete' flag is relevant because it ensures that every voice will make it through the entire system. Since the mixer will only signal completion after it has itself received EOF's from each processor, each C program has the ability to stall the entire system until it has fully completed its processing, guaranteeing functional correctness. Instead of an explicit EOF flag, it may be better to maintain a count of the total samples remaining in the mixer itself, and decrement that counter for each sample received. This setup would loosen the requirements for the C programs running on each processor, but requires knowledge of the total length of the voices beforehand. Since we don't need timestamps on samples due to the monotonicity of our hardware sample processing, the present delay for an EOF is fine. The contract with the SMIPS code is that it must check for EOF before fetching a sample and should not request sample information afterward. If this contract is broken, the C program may not complete. The SMIPS C interface includes a function Boole getSample(int* data) which will automatically know to check for EOF and return false if the EOF has been reached and there is no valid data. Sending the sample back to hardware is straightforward with no restrictions. Lastly, the SMIPS C program must flag when it is complete and the last sample has passed or else the entire process will fail to terminate.

Contributions

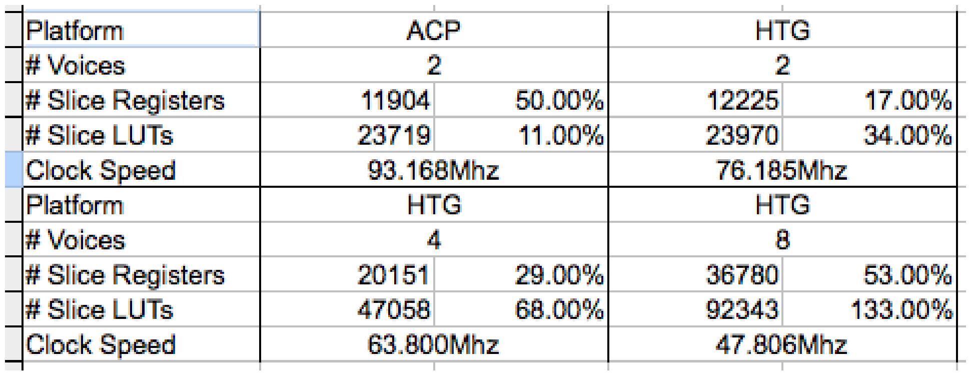
We have demonstrated the ability to implement the original goal of 12 plus voices with processing mixed together. The system has run successfully on the ACP platform with two SMIPS cores executing a simple pass thru SMIPS program that returns the original sample. We have also executed in sim using 16 voices and audio files of different lengths. The one successful FPGA compilation took 180 minutes while we've also seen four hours pass without the FPGA compilation finishing. This has definitely been problematic as the live work environment has not been tolerant to waiting for hours to verify the compilation works and not using the ACP host, courage in that time. Unfortunately, the one working build that successfully passed our short test set was accidentally reset before we had the chance to run proper meaningful audio through it. Things have gone well in sim though. In sim we can process different length audio files on up to 16 cores. Although the test bench code for both setting up the benchmarks and for host C++ code is fully automated and although the link between the between the host world and the FPGA isn't quite finished, both test bench and bluespec code are parametric making reconfiguring cores numbers easy. Thanks to the scratchpad, changing SMIPS programs is simple and fast too, not requiring any rebuild of bluespec. The drawback of being chained to sim is that running a meaningful test is exceedingly slow. Running a two voice test with 500 samples usually takes a couple minutes. We do have debug commentaries on but a 1

Algorithm 1 C interface for sample handling in SMIPS.

```
// the protocol for getting a sample is:
//check if eof, get
//the way to signal end is send last sample
//then flag progComp
void setComp( void )
{
  int set = 1;  //since I'm not sure how to do immediate...
#if HOST_DEBUG
  printf( "*** Program Completed ***\n" );
#endif
  asm("mtc0 %0, $26" : : "r" (set) );
}

// return if succeded or not.  If no success, then eof
bool getSample(int* data)
{
  int eof;
  //check if eof occurred- Note this also clears the flag
  asm("mfc0 %0, $25" : "=r" (eof) : );
  if (eof == 1)
  {
     *data = 0;
     return false;
  }
  //
  asm("mfc0 %0, $28" : "=r" (*data) : );
    return true;
}

//Send Sample to Hardware¿
void putSample(int data)
{
   asm("mtc0 %0, $27" : : "r" (data) );
}
```



Platform	ACP		HTG	
# Voices	2		2	
# Slice Registers	11904	50.00%	12225	17.00%
# Slice LUTs	23719	11.00%	23970	34.00%
Clock Speed	93.168Mhz		76.185Mhz	
Platform	HTG		HTG	
# Voices	4		8	
# Slice Registers	20151	29.00%	36780	53.00%
# Slice LUTs	47058	68.00%	92343	133.00%
Clock Speed	63.800Mhz		47.806Mhz	

Figure 5:

second sample is 44,100 samples. The time for a useful test does not scale well on sim. That said, if it works for 500 samples, it should work for 44,000. The other part of this is that we do have a handful of very simple SMIPS programs. They are thru or identity which simply returns the original sample, louder, softer, and bit-shifter. Louder acts as an amplifier such as mentioned in the mixer discussion. It is coded to mimic clipping as scaling a sample above the playable analog volume results in clipping rather than letting the number cycle over to something even more unhelpful. Softer is self explanatory. Bit shift does a bit reduction of the song from 16 bits to 8. Go Nintendo.

Synthesis Data

The slow part of the entire structure is the Mixer, which is slow because we naively used bluespec's sign extend function which is apparently very inefficient. Nevertheless, the entire system can operate fast enough to produce 44.1kH audio in real time. The HTG FPGA should support around 6 cores maximum. The ACP should be able to support around 20 cores. We don't have more data for the ACP because of the substantial time it takes to compile for that device ($>$ 9 hours).

Further Investigations

New Processor Architectures

Our efforts so far have focused primarily on basic operation and demonstration of principle. We have yet to get to the truly interesting architectures but we have achieved the foundation for implementing them. More powerful capabilities can be easily added. The first set of additional capabilities is to add full hardware processing with FFTs. This can be quickly implemented using the code from labs 1-4. As previously mentioned, the interfaces for the audio sam-

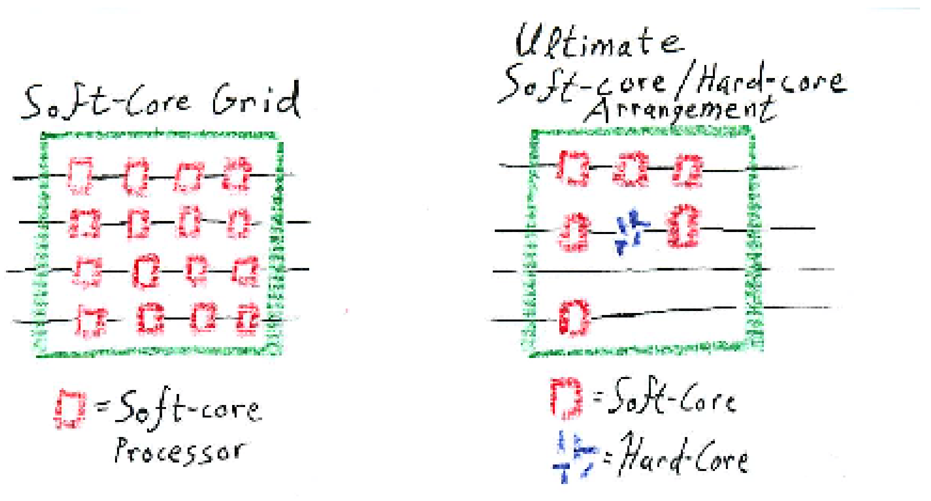
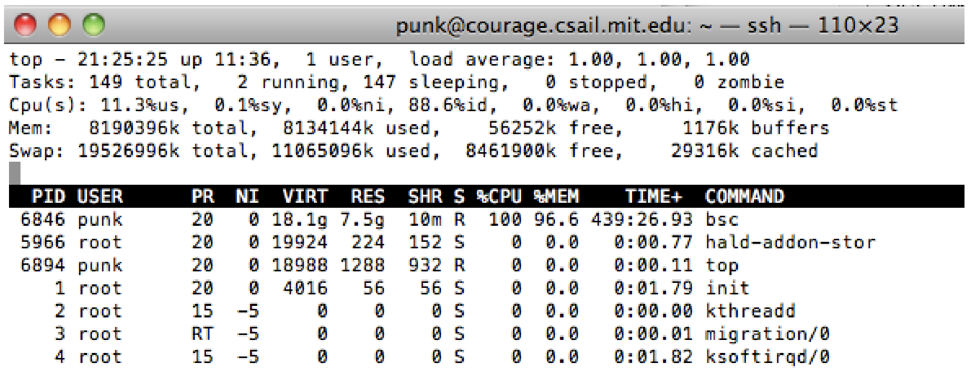


Figure 6: This dense processor grid is simple to reprogram. If a particular voice only needs a few processors, the extra processors can be loaded with "identity" programs which do nothing at all. To save space, only those processors which are actually needed should be stamped out on the FPGA. Also, hardware processing units such as FFTs can be easily interfaced with our system.

ple passing was derived from the FFT labs and is thus, easily adaptable. The only required change is to deal with the Invalids sent to fill pipes after EOF. Once this is complete, any of the existing FFTs can be plugged into the present pipeline system as a replacement for a soft-core. A second easily implementable interesting new capability is to build a soft-core with hardware acceleration of an audio FFT. This combines the benefits of the soft-core with the hardware FFT. This could be implemented by wrapping one of the present soft-cores in an FFT wrapper. Again, apart from the Invalids, the interfaces remain the same and again, we have the labs with code to compute the FFT. Yet another easy new implementation which is definitely targeted is to implement a matrix of cores. This enables the goal of running a single stream through multiple algorithms. If a stream does not need all the soft-cores in its path, those can be loaded with the existing pass-thru SMIPS code. It is less efficient but highly flexible and easy to code in BSV. Lastly there is the goal architecture where each pipeline has only the soft-cores (or FFTs) it needs. While this is extremely efficient, it is harder yet still reasonable to code. A definite goal in this implementation is an easy way to outline what the desired architecture is which the compilation code and then parse into the correct structures. We actually already have a routing table that would do a reasonable job of it, but we don't yet have the bluespec code the read the routing table and implement it.



```
top - 21:25:25 up 11:36,  1 user,  load average: 1.00, 1.00, 1.00
Tasks: 149 total,   2 running, 147 sleeping,   0 stopped,   0 zombie
Cpu(s): 11.3%us,  0.1%sy,  0.0%ni, 88.6%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8190396k total,  8134144k used,    56252k free,     1176k buffers
Swap: 19526996k total, 11065096k used,  8461900k free,    29316k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 6846 punk      20   0 18.1g 7.5g  10m R  100 96.6 439:26.93 bsc
 5966 root      20   0 19924  224  152 S    0  0.0   0:00.77 hald-addon-stor
 6894 punk      20   0 18988 1288  932 R    0  0.0   0:00.11 top
    1 root      20   0  4016   56   56 S    0  0.0   0:01.79 init
    2 root      15  -5     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S    0  0.0   0:00.01 migration/0
    4 root      15  -5     0    0    0 S    0  0.0   0:01.82 ksoftirqd/0
```

Figure 7: This is on courage.csail.mit.edu, with no other substantial programs running during the length of the compile.

Optimization

We haven't actually done any optimizations of the system yet. This is an obvious place for improvement. For instance, the mixer is presently the slowest stage as it is not at all internally pipelined. Splitting some of the math into separate pipeline stages would improve the performance speed. A good candidate would be separating the scalar multiplications from the other math. The processor itself can be further optimized with better branch prediction and other improvements. Some quick passes through the synthesis reports should highlight areas to target.

Development Issues

While our project was conceptually simple, many *trivial* things conspired to make it much more difficult than it ideally should have been. Some notes on creating a better system for designing hardware include:

- Better documentation for AWB.
- Less indirection within the AWB environment.
- Explicit licensing status. Failure to obtain a license should be a very noticeable error instead of the current tragedy of simply failing silently with no error message. You will fail any attempt to run a test after 12:00AM.
- More deterministic compilation. Figure 7 should not be possible.

Fun Bugs that we experienced:

- Some very simple rules will cause Bluespec to get stuck forever in a compilation loop unless `(* conservative_implicit_conditions *)` is on.

- Sending too much data to RRR will block it and cause unrelated sections of code also using RRR to fail.
- ScratchPad was 'updated' during the project, breaking our existing code.
- Xilinx does not assign the correct type of RAM blocks without extensive workarounds.

Getting closer to the target design

Three more significant places for improvement are implementing the clocking need for real-time live use of Pygar, switching the memory module over to being request based and specifically clocked along with the target of enabling dynamic control. Clocking the memory is a big step towards enabling live use of the design. Live audio flows at a specific sample rate and if it is to be read or the mixer is to drive real audio, any AD for audio in or DA for audio out will need to be clocked and thus send/receive one sample each clock. This type of functioning also impacts the desired memory design. With tighter timing deadlines, it makes more sense to request data rather than wait for it. This enforces the sample source to meet timing deadlines and failure to do so is easily identified. A typical problem in audio processing we have not yet discussed in detail is some of the peculiarities of reading multiple voices that thwart traditional memory access optimizations and cache schemes. Putting more memory fetch logic into a more true DMA module provides the opportunity to address some of these issues. It also means that the audio core can run largely independent of the need for any external processor. It makes it far more self-contained. The last big win of reasonable complexity is the addition of dynamic control. Right now we only support static settings. An SMIPs algorithm starts with a specified set of control settings and there is not yet a way to update these. The mixer also receives it's multiplication scalars through static elaboration. Clearly, being able to change the volume of song play back or re-balance voices is an important real use function. A possible architecture for the is to add a specialized control core that can pipe a control stream to each soft-core and mixer. Using a soft-core for this retains the flexibility to swap out plugins and update control software for this new plugin. Looking at what software can do, there are obviously still more way to add and improve functionality.

Our system right now represents only a fraction of the styles of Audio Processing enabled on FPGAs, but already it can outperform pure software systems. There are many other issues that need to be addressed, but already our system is useful.

Special Thanks to Eliott Fleming for his extensive help without which we would have not had gotten as far as we did.