Emulating a Combo Organ Using Faust

Sampo SAVOLAINEN
Foo-plugins
http://foo-plugins.googlecode.com/
sampo.savolainen@gmail.com

Abstract

This paper describes the working principles of a 40 year old transistor organ and how it is emulated with software. The emulation presented in this paper is open source and written in a functional language called Faust. The architecture of the organ proved to be challenging for Faust. The process of writing this emulation highlighted some of Faust's strengths and helped identify ways to improve the language.

Keywords

Faust, synthesis, emulation

1 Introduction

Faust[Yann Orlarey, 2009b] stands for Functional AUdio STream and as the name implies it is a functional language designed for audio processing. The Faust compiler is an intermediary compiler, which produces source code for a C++ signal processor class which is integrated into a chosen C++ architecture. This architecture file provides a run-time environment, or wrapper, for the processor. This wrapper can be for example a stand-alone Linux Jack application or an audio processing plug-in.

This paper describes how an emulation of a 1970's combo-organ was written in Faust. The Yamaha YC-20 is a fairly typical organ of its time, a transistor based relatively light instrument meant for musicians on the road. The organ is a divide-down design and its working principles are discussed in detail in section 2. The emulation is released as open source under the GNU General Public License (v3). In the spirit of open source, the working principles and decisions taken during writing the emulation are described in this paper for all to read.

The YC-20 was chosen to be emulated as its architecture is very different from typical software and virtual analog synthesizers. Instead of the complex controllability and routability of typical synthesizers, this organ requires a large number of fixed parallel processes and components. This makes for a good test of Faust's performance and parallelization capabilities. Access to an operational organ was also a factor in the choice, as it makes it possible to match the emulation quality against the original.

The contents of this paper is organized into four sections. The first section covers how the real organ operates. This is followed by a description of how the organ is emulated. The third section covers the performance aspects of the emulation. The last section offers analysis of Faust's strengths and gives proposals on how to improve the language.

2 YC-20

This section covers the operations of the organ in detail[Yamaha, 1969]. The information in this is section is later referred to and detailed further when discussing the emulated parts. Figure 1 shows a block diagram of the device.

2.1 Features

The organ has one physical manual with 61 keys with a switchable 17 key bass manual and two voice sections. Instead of drawbars like Hammond organs, the voices are controlled by a lever system used in Yamaha organs of the time. As the word drawbar is commonplace when discussing organ voicing, the word is used in place of lever for the voice controls. While the controls are potentiometers, they have notches to help the user achieve repeatable settings. Each drawbar lever in the organ has four positions (end stops and two notches) from off to full volume.

Section I has drawbars 16', 8', 4', 2-2/3', 2', 1-3/5' and 1' and section II has drawbars 16', 8', 4', 2' and a continuous brightness lever. The sections can be selected or mixed together using a continuous balance lever. Enabling the bass manual switches the lowest 17 keys (white on black) to bass section sounds with drawbars 16'

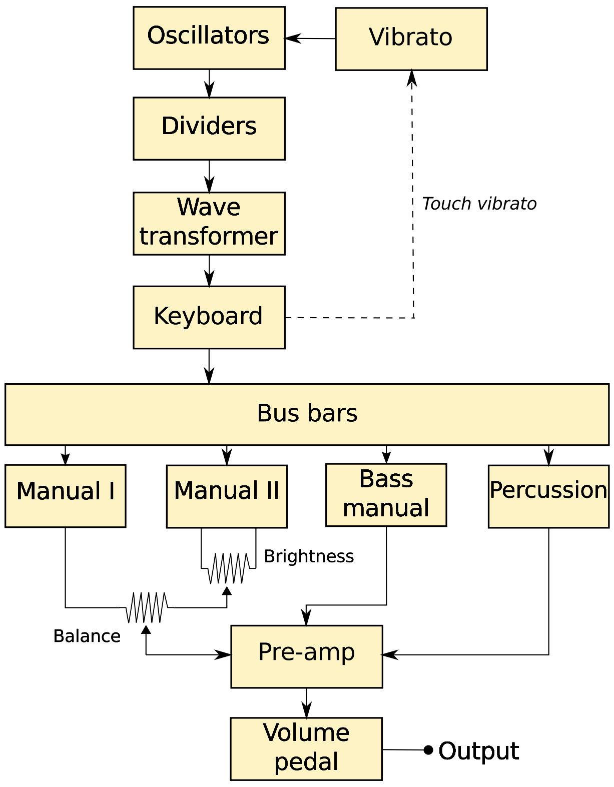

Figure 1: Block diagram of the YC-20

and 8'. There are controls for pitch, overall volume, bass volume, vibrato depth, vibrato speed and an obscure "touch vibrato" feature. The organ also has a single percussion drawbar and an integrated volume pedal. Like the drawbars, all vibrato controls have four notches.

2.2 Tone generation

The synthesizer architecture comprises of 12 oscillators, one per each note in a twelve-tone equal temperament (12-TET) scale. Each oscillator produces a sawtooth wave. For each oscillator, there are 7 frequency divider stages so each oscillator produces 8 voices totalling 96 voices. The dividers produce square waves.

These voices are next fed through a passive filter bank. The divided voices have both a high pass and low pass filters while the oscillator voices are only high pass filtered to remove bias. This totals 180 resistor-capacitor (RC) filter networks. Filtered voices are connected to switches on the actual keys of the keyboard. The keyboard is thus in fact a matrix mixer connecting the voices to bus bars. A key feeds each bus bar with the appropriate voice matching that key and octave, or the matching harmonic voice in case of the 2-2/3' and 1-3/5' bus bars.

2.3 Sections

The drawbar controls are potentiometers which control how much of each bus bar is mixed to each section of the organ. While in section I, drawbars are simply mixed together using resistors, Section II drawbars have more complex filtering. Each section II drawbar is divided into two streams, each filtered separately with RC networks. One stream is low pass filtered while the other is high pass filtered. The low passed and high passed signals are mixed through resistors to combine into bright and dull signals which are then buffered. The brightness potentiometer controls a mix of the two which, after further buffering, is the output of section II.

The key switches on the lowest 17 keys feed separate bass bus bars. The bass manual switch controls whether these bus bars are kept separate or mixed with the main bus bars. Thus, with the bass feature disabled, the bass keys work exactly as the rest of the manual. With the switch enabled, the bass bus bars are separate and only feed the bass section. It is worth noting that when enabled the bass section effectively discards the signal of bass bus bars 2-2/3', 2', 1-3/5' and 1'.

Bass section differs from the two main sections in that both bass manual drawbars (16' and 8') use a mix of multiple bass bus bars. The 16' drawbar is a mix of bass bus bars 16', 8' and 4'. The bus bars are mixed through different value resistors. 16' has the least resistance and 4' the highest resistance. The 8' drawbar is a mix of the 8' bass bus bar with lower resistance and 4' bass bus bar with higher resistance. While section I and section II drawbar control potentiometers are wired in a standard volume configuration, the bass manual drawbar controls are wired center-tap to source with the other tip grounded and the other tip as the output. This configuration makes the drawbar control not only affect how much of the drawbar is mixed in, but it also varies the impedance of the signal source. The different mixing resistors and the varying impedance of the drawbar controls, combined with a fixed capacitor to ground (after summing the drawbar signals) makes the network act as a fairly complex RC filter. This filter has a varying cutoff point and mix amount per bass bus bar.

2.4 Percussion

Percussion manual sounds are created by mixing together bus bars 1', 2-2/3' and 16' via resistors. There is a substantially larger resistance on the 1' bus bar leading to less 1' voices in the percussion section. The volume of this signal is controlled by a simple envelope generator. This envelope generator is triggered by activity on the 1' bus bar. The envelope attack is instantaneous and the release time is fast while the sustain level is zero. This causes a fast attack sound, but the effect only works when no keys are pressed down before. In other words, the percussion effect is not heard, for example, when playing legato. However there is a substantial amount of bleed from the percussion section which is audible when all other drawbars are off.

If the bass manual is enabled, it disengages the bass bus bars from the main bus bars. Thus the bass keys will not mix sounds onto the 1' bus bar. Therefore playing the bass manual will not trigger the percussion and percussion sounds will trigger even when bass keys are held down.

2.5 Main output

The main output of the device is a mix of sections I and II, the percussion part and the bass manual. The mix between sections I and II is controlled by the balance lever (a potentiometer). The amount of percussion mixed in is controlled by the percussion drawbar. Bass manual is summed into the main output via a potentiometer controlling the bass volume. This combined signal is then preamplified. The preamplified output can then be attenuated by the main volume potentiometer of the device and the volume pedal. The volume pedal action controls a mechanical shutter between a small light bulb and a light dependent resistor.

3 The emulation

The emphasis was on creating a playable instrument which sounds like the original organ. A playable emulation needs to be able to work at low latencies and it needs to be efficient enough to be ran on commodity hardware. The emulation tries not only to emulate the ideal circuit but also some of the inaccuracies in the real instrument.

3.1 Why Faust?

Instead of taking an approach where sounds are synthesized only when needed, this emulation keeps all oscillators, dividers and filters running all the time – just like the real organ. Faust was chosen as the programming language to emulate the organ with, as Faust's functional semantics fit well with having all processing running at all times. Faust also makes it trivial to have a large amount of streams flowing from one circuit to another. This emulation was also intended as a test of Faust in a real world use.

3.2 Tone generation

The 12 main oscillators produce sawtooth waves. They share a common, varying bias voltage (see section 3.3) which affects the oscillator frequencies. The main oscillator voices are divided by an array of flip flop circuits each dividing the frequency in half and the next one dividing the previously divided voice. The flip flops produce square wave signals. This means each oscillator produces a total of 8 phase-synchronized voices, each one an octave down from the previous voice.

As the main oscillator frequencies are high (4–8kHz), a naive oscillator implementation would suffer massively from aliasing when using typical sampling rates (44.1–96kHz). Naturally, also the dividers would suffer from aliasing as well. After evaluating different methods to band-limit the signals, the PolyBLEP[1][Välimäki and Huovilainen, 2007] method was chosen. While BLEP[Brandt, 2001] would produce less aliasing components, it is computationally more expensive. More importantly, the BLEP step function is currently impossible to calculate in Faust as it requires using Fourier and inverse Fourier transfer functions. The quality of BLIT-SWS[2][Stilson, 1996] is good at high frequencies, but it produces aliasing components below the fundamental frequency[Välimäki and Huovilainen, 2007]. Also, BLIT-SWS is problematic when it comes to band-limiting hard synchronized oscillators[Brandt, 2001] which is one strategy to emulate the divider circuits.

Second, third, and fourth order polynomial residual functions were evaluated for the PolyBLEP. The higher order residual functions reduced high frequency aliasing only slightly more compared to the second order function. Furthermore a band-limited signal using the second order function has considerably less aliasing components below the fundamental frequency when compared to the third and fourth or-

[1]BLEP = band-limited step function. PolyBLEP uses a step function derived from a simple polynomial

[2]BLIT = band-limited impulse train, SWS refers to the use of windowed sinc functions

der functions. This confirmed previous findings[Pekonen, 2007]. The chosen second order polynomial residual function *r(t)* is shown as equation 1.

$$r(t) = \begin{cases} t^2/2 + t + 1/2, & \text{if } -1 \leq t \leq 0 \\ -t^2/2 + t - 1/2, & \text{if } 0 < t \leq 1 \end{cases} \quad (1)$$

The divider circuits are emulated as slaved oscillators. The main oscillator function outputs both the signal and phase information. One divisor function divides the phase and feeds this to a slave oscillator. The slave again produces both a signal and phase information. The complete divider circuit for one oscillator is seven such divisor functions piggybacked.

PolyBLEP works by adding the polynomial band-limited step function to two samples: the sample before and after a discontinuity in the signal. To achieve this, the implementation delays its output by one sample. At each frame, the phase is inspected. If the phase passed a discontinuity in the waveform, the PolyBLEP function is evaluated for and added (rising wave) or subtracted (falling wave) for the previous sample and the current sample. As Faust lacks true branches, all possible branches of conditional statements are always calculated. Table 1 shows the amount of residual function evaluations required under different conditions. The table shows that branching would be far superior to any non-branching solution. This is because without branching the amount of PolyBLEP evaluations done is purely a function of the sampling rate while the amount of required evaluations is a function of the signal frequency.

To be able to run the emulation in real time, a truly branching solution had to be developed as an external C++ function. This was however easy to integrate with the Faust processing as the *ffunction* operator lets one use externals just as native functions.

The filter bank contains tailored filtering for each of the 96 voices produced by the oscillators and dividers. The main oscillators only have a single capacitor in series. This is emulated as a high-pass RC filter using an approximation of the next stage impedance as the load resistor value. The first four divided signals have a trivial RC low-pass filter before a single capacitor in series similar to the the main oscillator filter. The lowest three voices are filtered like the previous four, except that there is a resistor to ground after the series capacitor. This alters the next stage impedance compared to the other filters.

Frequency	(A)	(B)	(C)
5kHz	352 800	44 100	10 000
1kHz	352 800	44 100	2 000
500Hz	352 800	44 100	1 000

Table 1: Amount of PolyBLEP calculations per second for a square wave at Fs = 44.1kHz. (A) an implementation where per each frame the PolyBLEP is evaluated for both the previous and current sample for both discontinuities. The branched nature of the residual function multiplies this number further by a factor of two. (B) an ideal implementation without branching where only one PolyBLEP is evaluated per frame. (C) is the number of required PolyBLEP calculations.

Filtering is very complex to emulate exactly right, as one voice might be connected to multiple bus bars. This varies the high-pass filter load resistance and therefore affects the filter cutoff frequency depending on what keys are depressed. This is not emulated. Instead, the high-pass filter load is estimated based on the expectation that there is only one connection to a bus bar.

3.3 Oscillator bias

Each oscillator produces a saw wave at a different frequency. The frequencies are chosen from a 12-TET scale. However all oscillators share a single bias voltage affecting their frequency. This bias voltage is controlled by the vibrato circuit, touch vibrato and the master pitch potentiometer. The master pitch potentiometer is emulated by a simple slider widget with a DC output. Touch vibrato would be controlled by horizontal movement of the keys. As such MIDI keyboards are extremely rare[3] this feature was left out of the emulation.

Vibrato control voltage is created by a simple sine wave oscillator. The vibrato speed controls the speed of this oscillator. The vibrato speed range (5-8Hz) was measured from the real device. The vibrato depth is simply an attenuation control for the vibrato oscillator output. Vibrato depth range was empirically selected. The depth control deliberately never goes to

[3]Only one keyboard was found claiming such capability: the Yamaha STAGEA ELS-01C.

zero, thus the vibrato has a miniscule effect on the bias voltage even when turned down.

3.4 Keyboard matrix mixer

The keyboard matrix mixing, while a relatively simple part, contains many separate operations. Each key is a Faust button[4] connecting multiple voices to different bus bars. This means each key button is used as a multiplier for 7 different voices (one per bus bar). So the signals on the emulated bus bars are the outcome of $61 * 7 = 427$ discrete multiplications summed to appropriate busses. There is also added logic for the bass manual where the resulting downmixes from the 17 bottom keys are fed to either the main bus bars or the bass bus bars.

The matrix mixer could be written as sets of floating point tables multiplied together. In Faust however, there are no such array operators. This results in a number of separate multiplications and sums which are difficult for the Faust compiler to optimize, as vector operations are unusable if the data is not in ordered arrays.

The key action is not band-limited. The key switches on the real organ are simple switches connecting voices to bus bars. This operation causes clicks in the real organ as switches open and close. The naive non-band-limited key action of the emulation matches the sound of the real organ surprisingly well. However, this is something that could be improved at a later stage.

Writing the keyboard matrix mixer also revealed an issue with the service manual. The service manual indicates wrongly which voices are connected to the harmonic bus bars (2-2/3' and 1-3/5'). The manual states that 2-2/3' is connected to a voice five semitones higher than the voice connected to 4' and 1-3/5' with a voice eight semitones higher than 2'. The real organ connects voices seven and three semitones above respectively. The emulation follows the real organ instead of the service manual.

3.5 Section I

Section I is a mix of the main bus bars mixed together through the drawbar potentiometers and additional resistors. There is no extra filtering applied to the signal. Thus the only part left to emulate for section I sounds is the drawbar controls.

The drawbar potentiometers do not follow the typical linear or logarithmic tapers. The real organ was measured while playing a stable note with all drawbars off except one drawbar which was tested in all three on positions. The off position is expected to be -inf. Compared to full volume, the two middle positions were measured at -12dB and -18dB.

[4]Button output signal is 1 when it is being pressed down and otherwise 0.

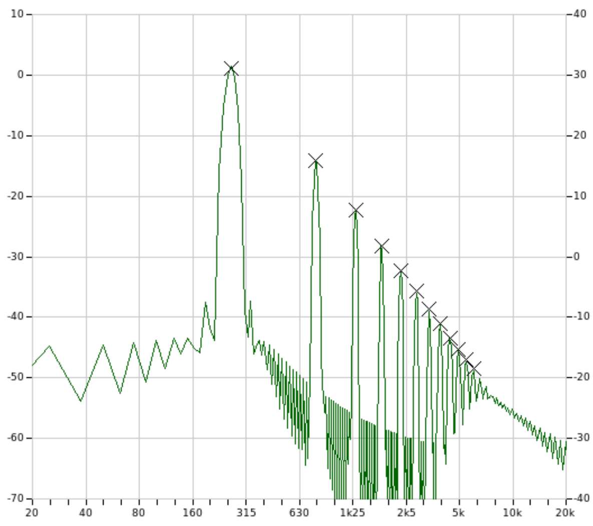

Figure 2: Comparison between emulated and real manual I drawbar 8' sounds.

This can be translated into a continuous transfer function (see equation 2). p is the position of the lever from 0 (off) to 1 (full volume). The function returns a gain coefficient usable in the emulation. The function emulates the taper and can be used with slider controls provided by the Faust architecture. Further work on a graphical user interface should provide four position levers.

$$\begin{aligned} \mathrm{coeff}(p) &= 2.81p^3 - 2.81p^2 + p \\ \mathrm{coeff}(0) &= -\mathrm{inf\ dB} \\ \mathrm{coeff}(1/3) &\approx -18.05\ \mathrm{dB} \\ \mathrm{coeff}(2/3) &\approx -12.03\ \mathrm{dB} \\ \mathrm{coeff}(1) &= +0.00\ \mathrm{dB} \end{aligned} \tag{2}$$

Figure 2 shows C2 played on section I with only the 8' drawbar engaged. The line depicts the frequency spectrum of the emulation output. The crosses show peaks measured from a YC-20 organ using the same settings.

3.6 Section II

As described earlier in section 2.3, Section II has a controllable brightness feature. The variable brightness is done by dividing each bus bar into two streams, one of which is high-passed and the other low-passed. The streams derived from different bus bars are then mixed together into bright and dull streams. The brightness control is a simple balance control between the two streams.

The high-pass filtering is done by a two stage RC filter with a resistor in parallel with the capacitor of the first filter. Thus the filter is effectively a shelving high pass filter. The shelf is emulated by calculating a voltage divider between the parallel resistor and the resistor in the RC high pass filter. The voltage divider gives a gain coefficient C which is used to mix together the high-passed and unfiltered signals. Unfiltered signal is multiplied by C while the filtered is multiplied by *1-C* and the results are added together. This keeps gain at high frequencies at 0dB as with the original passive filter. The two passive filters also affect each other and could not be emulated by simply chaining two digital filters. The load applied by two filter stages is emulated by dividing the resistance of the second filter by two. This doubles the cutoff frequency of the filter. This method was found by trial and error. Figure 4 shows the block diagram of the complete filter.

The low-pass stage is much more straightforward. Filtering consists of two chained RC low-pass filters. However, the best match with the original organ was achieved by not compensating for load posed by the chained filters. This filter stage is emulated by two digital RC low pass filters in series.

Figure 3 shows C2 played on a fully bright section II with only the 8' drawbar engaged. This measurement was done at the same overall level as measurements in figure 2. The measurement shows a slight overall volume difference and that the section II high-pass filters are not perfect. However, while harmonics for this particular sound do not line up exactly, the balance between harmonics is good enough. The harmonics of many other notes (for example C3) line up perfectly.

3.7 Percussion

The percussion uses the signal on the 1' bus bar to trigger an envelope generator. There is a root-mean-square envelope follower on the 1' bus bar and the envelope is triggered when the follower exceeds a set threshold. The envelope generator starts off with a signal where a one frame unit impulse is created at the trigger point. This impulse signal is low-pass filtered to match the measured percussion envelope. This

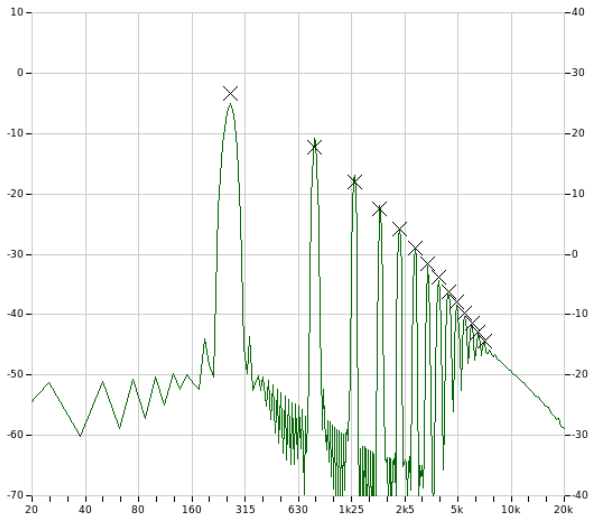

Figure 3: Comparison between emulated and real bright section II drawbar 8' sounds.

envelope is then used to control the volume of a mix of the 1', 2-2/3' and 16' bus bars. Some bleed is added to the envelope generator output to emulate the bleed exhibited by the real organ.

3.8 Bass manual

Emulating the bass section perfectly is a complex task due to how mixing resistors and the variable impedances of the potentiometers affect the RC filter. Measurements from the real organ however suggest that the low frequency fundamentals dominate the signal and an acceptable emulation is achieved by carefully controlling the mix of the bass bus bars and by using a single fixed RC low-pass filter. This one filter is applied to a mix of both bass drawbars.

4 Performance

Running this emulation requires a substantial amount of processing power. There are 96 oscillators working all of the time and there are hundreds of filters being applied at many stages of the design. Faust is designed to easily harness multiple processing units in parallel. The design of the emulation has a lot of potential for parallelization and as such, it should work as a good real world test for Faust's SMP features.

The performance numbers shown here are measured using a modified version of the Faust benchmarking suite. These numbers are estimates of achieved memory bandwidth in megabytes of audio data generated per second. The Faust benchmark uses this measurement, as memory bandwidth on SMP systems is a precious commodity. In the case of the YC-20, the processing creates much more data than what comes out of its single mono output. Thus these figures can only be compared to each other.

Tests were done on a Dell D820 laptop with a Core 2 Duo T7400 (2.16GHz) processor using the internal sound interface. The operating system was 32 bit Ubuntu 9.10. The processor frequency governor was switched to performance on both cores before running the tests. The kernel was a standard Ubuntu package, 2.6.31-19-generic and the used gcc version was 4.4.1-4ubuntu9. Benchmarks were done on YC-20 code revision 227[5]. Figures in column C are from a slightly modified version of the YC-20 code and it shows the performance impact of non-branching PolyBLEP calculations. The results in MB/s are shown in table 2.

Faust benchmark	(A)	(B)	(C)
scalar (galsascal)	0.40	0.43	0.28
vectorized (galsavec)	0.58	0.59	0.35
vectorized 2 (galsavec2)	0.62	0.62	0.34
OpenMP (galsomp2)	0.51	0.55	0.29
scheduler (galsasch)	0.90	0.88	N/A
scheduler 2 (galsasch2)	0.93	0.89	N/A

Table 2: The tests were ran with different sets of gcc parameters:
(A) Branching C++ PolyBLEP evaluations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize
(B) Branching C++ PolyBLEP evaluations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize -fgcse-sm -funsafe-math-optimizations
(C) Divider slave oscillators use non-branched PolyBLEP operations. gcc -O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize -fgcse-sm -funsafe-math-optimizations
GCC was unable to compile the scheduler versions of the emulation with non-branching PolyBLEP operations.

The multi-threaded scheduler tests were notably faster than other benchmarks. There is an increase of 50% in memory bandwidth when compared to vectorized single thread tests. This data is backed by tests done by measuring the performance of the YC-20 emulation running as a jack application. The measurements shown in table 3 were done by running the emulation compiled with different flags and observing the DSP percentage meter in qjackctl. This meter tells how much of the time between jack engine callbacks is spent doing processing inside jack clients.

Tests	Faust flags	Load
scalar	none	$\sim$ 49%
vectorized	-vec -vs 32	$\sim$ 34%
scheduler	-sch -g -vs 256	$\sim$ 29%

Table 3: All tests were compiled with the following gcc flags:
-O3 -march=native -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize.
Jack 1 was running real-time with a dummy back-end simulating a 48kHz sample rate with 512 frame buffers.

5 Conclusions

The software presented in this paper emulates the YC-20 organ well. While some parts of the emulation could benefit from more polish, other parts of the emulation can sound very much like a real YC-20. A better quality anti-aliasing method could be beneficial, although these methods would come at the expense of CPU cycles. The real organ has considerable bleed and inconsistencies which might also be worth emulating.

5.1 How to improve Faust

As discussed in section 3.2, and shown in the benchmarks in section 4, there is a strong case for a truly branching select operation. However, the semantics of Faust requires all of the processing graph to be evaluated for every frame. Truly branching operations would cause some function evaluations to be skipped. The operating semantics are however compatible with branching operations if the skipped functions are stateless. This is because if the function has no state, the next evaluation of the function does not depend on previous evaluations.

There are two ways truly branching operations could be added to Faust: either select2 and select3 calls with no stateful function branches are compiled automatically as truly branching or a new branching select call is created. Both cases require the specification of rules on what a stateless function is. Also, the compiler must

[5]Subversion repository available at http://foo-plugins.googlecode.com/svn/trunk/

check whether the functions being skipped over comply to these specifications before allowing compilation of such code.

This logic could also be generalized to multiplication functions. Function $f(x)$ in equation 3 is an example of a function where evaluating $t2(x)$ can be skipped if $x \leq 0$ as $t1(x)$ evaluates to 0 for $x \leq 0$ and $t2$ is stateless.

$$\begin{aligned} t1(x) &= (x > 0) \\ t2(x) &= x^2 - fmod(x, 1.0) \\ f(x) &= t1(x) * t2(x) \end{aligned} \tag{3}$$

The compiler error messages are often unhelpful. They do not always specify in which file the compilation error occurred. Optimally the compiler should identify the file name, line number and if possible, the name of the function the error occurred in. One specific issue with the compiler messages is worth giving special attention to. It is cases where the number of inputs and outputs of functions in a sequential composition do not match. In such cases, instead of printing the names of the offending functions, Faust outputs a quite exhaustive description of both functions. For example, if the YC-20 emulation would fail to cut the phase output of the last divider (dividers would then output a total of 108 instead of 96 streams), the resulting error message is 67 kilobytes.

For signal processing, the lack of support for Fourier transforms restricts what problems can be solved. As mentioned in section 3.2, the better band-limiting method (BLEP) is not possible to implement in pure Faust. Fourier transforms would naturally be useful for a variety of other tasks as well. Fortunately there has been promising news of multi-rate support for Faust which would allow processing such as Fourier transforms.

As procedural programming is the prevailing model of programming, it is safe to say many potential users of Faust are familiar with only that model. This applies pressure to the quality of documentation. The current documentation should abbreviate the definitions of the more advanced features of the language. Especially recursion and the rdtable and rwtable functions could benefit from better explanations. Currently the features are almost side-stepped and almost nothing but their syntax is described.

Implementing the matrix mixer required a lot of hand-crafting. Having a concept of indexable arrays would help such operations. If the 96 inputs of the keyboard mixer and the buttons representing the keys of the organ could be arranged into arrays, the mixing could be done algorithmically using the aggregate functions such as sum and vector multiply. Such operations also open up ways to further optimize the created C++ code. As it stands, the keyboard mixing is compiled into a large amount of separate discrete multiplication and addition operations which can not be vectorized.

The Faust code shown as equation 4 is ambiguous, but can be compiled without any warning. This can lead to anywhere from unexpected results to complete failure. It would be good form for the Faust compiler to at least warn the user of the situation.

$$\begin{aligned} f(x, y) &= x + y \\ with\{x &= y * y;\}; \end{aligned} \tag{4}$$

5.2 Benefits of Faust

Functional thinking suits audio applications very well as solutions to problems can often be expressed as mathematical equations. This model sidesteps many of the issues procedural programs face, as the processes can be written at a higher abstraction level. Unlike with procedural languages, Faust allows signal processing programmers to not worry about buffers, their lengths or loop structures or real-time thread constraints. This also saves time for the programmer as less time is spent debugging issues not directly connected with the signal processing task at hand.

Faust code is easy to keep readable as the syntax offers tools, such as sequential composition, which help keep functions simple and concise. See listings 1, 2 and 3 for an example. While listing 2 is pretty compact, one should note that functions are presented to the reader in reverse order and it is also difficult to see to which function return value the multiplication is applied to.

```
float dostuff(float f) {
  f = func1(f);
  f = func2(f);
  f = f * 2.5;
  f = func3(f);
  return f;
}
```

Listing 1: Consecutive calls

```
float dostuff(float f) {
  return func3( func2( func1(f)) * 2.5 );
```

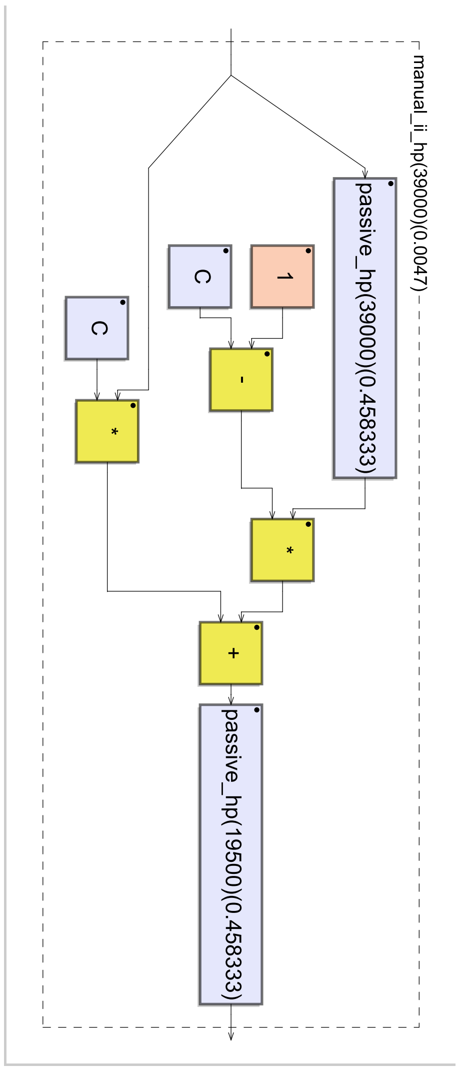
manual_ii_hp(39000)(0.0047)
passive_hp(39000)(0.458333)
C
1
C
-
*
*
+
passive_hp(19500)(0.458333)

Figure 4: Faust SVG output of section II high pass filtering for a single drawbar. The parameters for the filters are resistance in ohms and capacitance in microfarads.

```
}
```

Listing 2: Enclosed statements

```
dostuff = func1 : func2 : *(2.5) : func3;
```

Listing 3: Faust sequential composition

The SVG output of the Faust compiler is also worth mentioning. The compiler can create hyper-linked SVG documents which depict the process function. This document can be an excellent learning and debugging tool as it shows how the compiler understood the source code. Figure 4 shows an example of this output.

The stream concept also makes code reusable. When combining procedural code from multiple sources you often need to either convert data types between the different modules or refactor the modules to match. Faust processors naturally fit with each other. This model works very well with open source as it makes it easier to spread good ideas and implementations. It might even lead to a resource library of truly reusable components usable with any Faust project – as long as the licenses are compatible.

Faust also provides automatic parallelization and vectorization. This allows all Faust programs to benefit from these advanced optimization methods which usually require expert programming skills to implement. These capabilities have been discussed in detail by Orlarey, Fober and Letz[Yann Orlarey, 2009a]. Faust also allows the developer to easily test and compare different optimization methods without refactoring their code.

6 Acknowledgements

Thanks to Petri Junno for support and all the help with analyzing the original organ. I would also like to thank Sakari Bergen, Torben Hohn and Yann Orlarey for providing advice. Thanks also to Fons Adriaensen for Japa which was an excellent tool for comparing the emulation to the real organ.

References

Eli Brandt. 2001. Hard sync without aliasing. In *Proceedings of the International Computer Music Conference (ICMC)*, Havana, Cuba, September.

Jussi Pekonen. 2007. Computationally efficient music synthesis – methods and sound design. Master's thesis, TKK Helsinki University of Technology.

Tim Stilson. 1996. Alias-free digital synthesis of classic analog waveforms.

Vesa Välimäki and Antti Huovilainen. 2007. Antialiasing oscillators in subtractive synthesis. *Signal Processing Magazine, IEEE*, 24(2):116–125.

Yamaha, 1969. *YC-20 Service Manual.* Yamaha Corporation.

Stephane Letz Yann Orlarey, Dominique Fober. 2009a. Adding automatic parallelization to faust. In Grame, editor, *Linux Audio Conference 2009.*

Stephane Letz Yann Orlarey, Dominique Fober. 2009b. *FAUST : an Efficient Functional Approach to DSP Programming.*