Arco: A Flexible Audio Processing Framework

Roger B. Dannenberg
Carnegie Mellon University
rbd@cs.cmu.edu

ABSTRACT

Arco is a sound analysis, processing, and synthesis framework. Arco is small, configurable and language-agnostic, offering a modular approach to music systems building as opposed to monolithic systems offering a more encompassing but less flexible solution. Arco can be configured as a small library with only the code needed for a particular application. Arco builds upon O2, an OSC-like message system, allowing Arco to operate as a separate server process or thread, avoiding synchronization problems between stringent Arco signal processing timing and less predictable control applications. The design of Arco unit generators is described in detail and supported by benchmarking studies to evaluate the cost of each feature. To avoid introducing yet another unit generator library, Arco leverages FAUST and its library for algorithms which are translated automatically to be compatible with Arco. Arco also includes some interesting analysis functions for pitch, RMS and onset detection.

1. INTRODUCTION

Many different libraries, languages and applications exist for music audio signal processing, including CSound [1], Nyquist [2], Supercollider [3], ChucK [4], Pd [5], and Max [6]. These are large integrated systems that include both DSP functions and programming languages to control them. Lower-level frameworks and libraries exist, such as STK [7], JUCE [8], q [9], CLAM [10], and FAUST [11]. These facilitate custom DSP design as opposed to assembling ready-made modules (unit generators) but generally expose lower-level representations available through C++ and leave run-time organization (aside from the loop or callback that computes audio) up to the programmer. FAUST is an exception where DSP algorithms are expressed in a very high-level functional language that is compiled to a variety of target languages and plug-in standards.

There is an intermediate area where one wants to extend an application program with sound capabilities. A common solution is to communicate control information to a computer music system such as Max or Supercollider, but these are heavyweight solutions that may require extra time and effort to start and configure. These large programs cannot be embedded in other programs or run on embedded systems such as microcontrollers.

One effort to create a simpler subsystem for music synthesis is libpd, which is a library version of Pd, but libpd runs patches created with Pd and thus includes Pd code for loading and interpreting patch files as well as a complete set of all "standard" Pd functions. Another option is to create a DSP implementation in C++ using FAUST or RNBO (rnbo.cycling74.com/). This results in a fixed program for signal processing as opposed to a more dynamic library where one can alter the configuration while running.

Arco is a new framework intended to offer a lightweight library or server for audio DSP that can be used to construct real-time music audio applications and compositions. More like FAUST or RNBO, it has a small footprint suitable for linking into a larger application or running in an embedded system such as a Raspberry Pi (www.raspberrypi.com/). Arco is also like libpd or Supercollider's DSP server scserve in that users can instantiate and interconnect software DSP modules on-the-fly.

Arco is designed in a modular way so that it can function in different ways as (1) a minimal library controlled by almost any language, (2) a stand-alone server process that other applications can control, and (3) a scriptable application development system with its own high-level language.

As a minimal library, Arco uses a *manifest* file to specify a set of unit generators to make available at run time. Programs use the library by sending messages to create and interconnect unit generators. Additional messages can change parameters and reconfigure unit generators in real time. As a server, Arco can be compared to scsynth from Supercollider, i.e., a DSP server without the front-end programming language and interface. The server version has a small "main" program, to send and receive O2 messages, linked with an Arco library. As an application development system, Arco offers a full programming language as the "front end" and an Arco library as the "back end." It can be compared to Supercollider or ChucK. Some of the differences are:

1. Arco applications are compact statically linked programs with minimal external dependencies,
2. Arco's optional scripting language, Serpent [12], uses a Python-like syntax that some may find more flexible that ChucK and easier to learn than sclang (and a pure Python implementation is in the works),
3. Arco can leverage FAUST to dynamically adapt to multichannel signals (see below).

Published as: Roger B. Dannenberg, "Arco: A Flexible Audio Processing Framework," in *Proceedings of the International Computer Music Conference,* International Computer Music Association, 2025.

The basic architecture of Arco is described in the following section. Section 3 describes Arco unit generators, and Section 4 presents an analysis of various design options in terms of performance. Section 5 describes the software interface and control of Arco. Section 6 describes how FAUST can be used to create new unit generators for Arco, and Section 7 describes some applications. Finally, Section 8 presents a summary and conclusions.

2. SYSTEM ARCHITECTURE

Arco is designed using multiple abstractions to support different levels of functionality. At the lowest level, the "glue" that connects components is O2, a message system based on Open Sound Control (OSC) [13]. O2 provides message passing between threads sharing memory, between processes on the same computer, across local area networks, and even across the internet. Discovery mechanisms allow O2 processes to interconnect without configuring IP addresses or port numbers. For communication with Arco as a shared library, the complete CPU time to compose and deliver a message is about 320 ns.

Audio processing in Arco runs on a single thread, typically as the callback handling a PortAudio stream. This audio thread shares memory with an "application" thread running the full O2 library, which potentially communicates over TCP/IP sockets with clients.

When Arco runs as a server process (see Figure 1, top), the main server thread does little more than forward messages from sockets through shared memory to the audio thread and back. The server uses a minimal text-based interface. One use of the server is to provide audio processing services to a Python application, using O2 to communicate.

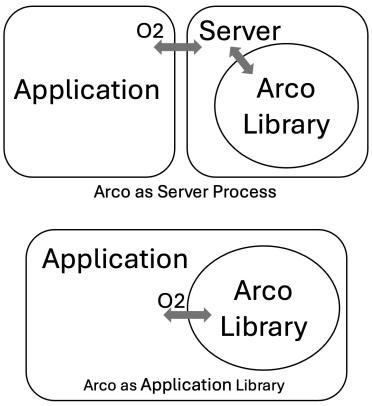

Figure 1. Arco can run as a server process that communicates with a separate application process using O2 messages (top), or Arco can be linked directly with an application or scripting language to extend it with audio processing. In both cases, O2 is used for control.

Arco can also be linked directly with a more complex application (see Figure 1, bottom), in which case the application polls the O2 library to send and receive messages. In between polling, the application thread can perform tasks such as computing control information for audio processes and running a graphical user interface. One such application is the Python-inspired Serpent programming language, a real-time object-oriented language with which one can write Arco applications, complete with a graphical interface, using a Python-like syntax [12].

3. THE UNIT GENERATOR MODEL

A fundamental building block in most audio DSP frameworks is the unit generator. A unit generator typically implements a simple DSP operation such as a digital oscillator, gain control, a filter, or signal adder (mixer). Unit generators are patched together to form a directed graph that accomplishes a complex analysis or synthesis process. Arco's unit generator model is designed to:

- Efficiently support low-latency interactive audio,
- Allow signals to have multiple channels,
- Process a mix of control-rate and audio-rate signals as well as parameters that can be updated by the application,
- Implement signal termination so that resources can be freed automatically at the end of sound events.

As in most audio DSP systems, Arco uses unit generators to process audio synchronously in blocks. In practice, audio I/O is done in blocks because the underlying operating system cannot operate at such a low latency or high rate of process scheduling. Therefore, except where there are feedback loops in the DSP, there is no advantage (other than simplicity, as in STK and ChucK) to sample-at-a-time processing. Arco uses 32-sample blocks by default, but the number can be changed at compile time. The following subsections describe other details of the Arco design.

3.1 Block-Rate Processing

While Arco processes audio-rate signals one block at a time, Arco also supports block-rate signals consisting of one sample per block (also known as control-rate or k-rate signals in other systems). In other words, the block rate is the audio rate divided by block size, conventionally $44{,}100 / 32 \approx 1378$ Hz. Since control is often more complex than basic signal processing, there may be more block-rate than audio-rate unit generators, so supporting block-rate computation can provide a great increase in efficiency. However, when block-rate signals are combined with audio-rate signals, discontinuities at the block rate can lead to audio distortion. Most Arco unit generators that operate at audio rates up-sample block-rate input signals using linear interpolation to eliminate audible distortion. When this up-sampling is done in the audio-rate inner loop, it amounts to just one register-to-register add per sample, so the computational cost is minimal.

3.2 Mixed Block-Rate and Audio-Rate Inputs

Unit generator inputs can be block-rate signals, audio-rate signals, or "constants" that can be updated by messages. Decoding these input types in the inner sample loop would introduce too much overhead, so Arco uses multiple implementations of each unit generator, each specialized for a different combination of block and audio rate inputs. In general, if there are n input signals, the unit generator implements 2^n functions to cover all combinations of block-rate and sample-rate inputs. Typically, $n \leq 3$, so the code size is small. In problem cases, a unit generator can

"coerce" an input to be one rate or the other by instantiating a resampling unit generator. This is less efficient but reduces the code size when there are many inputs.

3.3 Multi-Channel Signals

Since multi-channel processing is common, it is convenient to have some representation of multiple-channel signals so that the user does not have to replicate signal processing graphs to handle multiple channels. Nyquist introduced the idea of arrays of signals to represent multi-channel sounds and their automatic expansion into arrays of unit generators, an idea that was adopted by Supercollider. In Arco, we go one step further by supporting arrays directly in unit generator implementations (see Figure 2).

The default in Arco is to set the number of output channels to the maximum channel count of any input. Single channel inputs are virtually duplicated or "expanded" into multiple channels. E.g., if you multiply a stereo audio signal by a single-channel gain, the gain signal is duplicated and applied to both channels. Similar behavior allows control parameters to be duplicated across multi-channel filters and other effects.

Input connections can change, but a new input channel count does not propagate to outputs. (Otherwise, an input change could ripple through an entire patch, demanding a burst of computation and causing confusing behavior.) Inputs must be either single-channel or match the output count. If this condition is not met, only channel 1 is used.

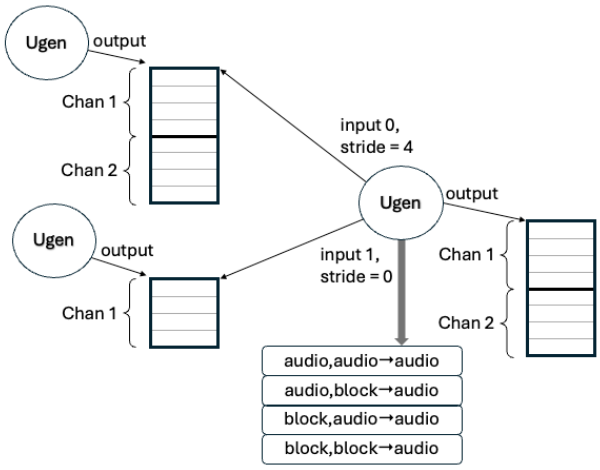

Figure 2. Arco unit generators can process multiple channels. Here, input 0 has 2 channels and input 1 has 1 channel. Since both inputs are audio rate, the audio,audio→audio channel process function is invoked once for each output channel. After computing channel 0, the input pointers are advanced by their stride, so input 0 will take samples from channel 2 while input 1 (stride = 0) will re-read samples from channel 0. The figure shows 4-sample blocks for convenience, but the default size is 32.

To explain how we access multi-channel inputs, first note that output is represented by a contiguous memory area filled with a block of channel 0 samples followed by channel 1, 2, etc. Also, when executing a unit generator, we invoke the unit generator DSP code in a loop that iterates to compute each output channel. The "trick" to managing a variety of channel counts is to associate a *stride* with each input that is added to a pointer that indexes the input signal (which of course is the output of some other unit generator). If the input is single-channel, the stride is zero and the same samples are re-used for each channel. If the input is multi-channel, the stride is the block size: 1 for block-rate, and the block size (32) for audio-rate inputs. Thus, one inner loop can handle any mix of single- or *n*-channel input signals.

Parameters that change only when updated by control messages are represented by a special Const unit generator whose output is normally fixed and which has no associated per-block computation. A Const value must still be accessed as if it is a block-rate signal, but the overhead is small. We also considered treating "constants" as a third input type, but the general case would require 3^n specialized functions rather than 2^n for only a small gain in efficiency.

To summarize, Arco unit generators can process any number of channels, and the number is set when the unit generator is instantiated. Each input can be single channel, which fans out to *n* channels, or *n* channels, and inputs can be audio rate or control rate. Each combination of audio and control rate is implemented by a separate specialized method. Reconnections and reconfigurations can occur between block computations, and the flexibility minimizes the chance that a user will make a nonsense connection that causes unwanted behavior or complete failure. Not all unit generators follow this standard pattern, e.g., the `stdistr` unit generator distributes *n* inputs across a stereo field.

3.4 Termination

Since Arco allows unit generators to be created and reconnected on-the-fly, it is possible to instantiate a new subgraph for every new sound. E.g., a MIDI note-on could be implemented by constructing a graph to synthesize that single note instance. This is generally simpler than managing a set of fixed resources. However, an interesting problem with the *instance* approach is how to free the instance. Usually, a gain envelope provides a smooth ending to a sound event, but we need some way to associate that ending with a set of unit generators that are no longer needed.

One way to free a sub-graph is to arrange for a message to be sent when an envelope ends. A message handler can then free a set of unit generators. Arco supports this approach, but it has problems: In particular, when graphs are nested and the outer graph completes first, the inner graph may not get a chance to complete, leaving an "orphan" handler that never gets a completion message. Since the orphaned handler has references to unit generators to be freed, they may be retained forever.

An alternative method is to add a "termination" state to unit generators that indicates the output will remain at zero permanently. The termination state can propagate through the graph along with audio. E.g., if an input to a multiplier terminates, then the multiplier output will become zero, so the multiplier terminates and the other input can be freed. When termination propagates to a mixer input, the input can be removed, the reference freed, and an entire subgraph providing that signal can be freed automatically. An example from the programming point of view can be found in Section 5.2.

Note: There was a bug in the benchmark. After correcting the problem, the actual computation time reduction when multiple channel support is removed is 4.5%.

3.5 Order of Computation

Unit generators are typically run in order from audio input to audio output to minimize the propagation latency from input samples entering the graph to output samples leaving the graph. In contrast, some systems evaluate unit generators in the order they are defined or invoke them procedurally in a user-defined order. This is problematic when graphs are dynamic because making new connections in the graph can change data dependencies that imply re-ordering the computation. Arco solves this problem by treating the graph as a data-flow network and performing a topological sort *for every graph traversal*, bringing signal sources up to date (recursively) before they are consumed.

4. PERFORMANCE MEASUREMENTS

How do these design decisions impact performance? Using a synthesis configuration from [14] (see Figure 3), I created a benchmark by combining Arco unit generator code into a non-real-time program that computes one hour of audio as fast as possible. Variations of this program were measured to assess the impact of different design decisions.

The run time for the "plain" Arco unit generators and 32-sample block size is 0.980 s. All measurements were performed 25 times on a 2022 Apple MacBook Air and the average time is reported. The standard deviation of run times is less than 1% of the average in all cases. It is interesting that this measurement represents an over 540x speed increase over the 25MHz RS6000 machine running essentially the same computation in 1997 [14]!

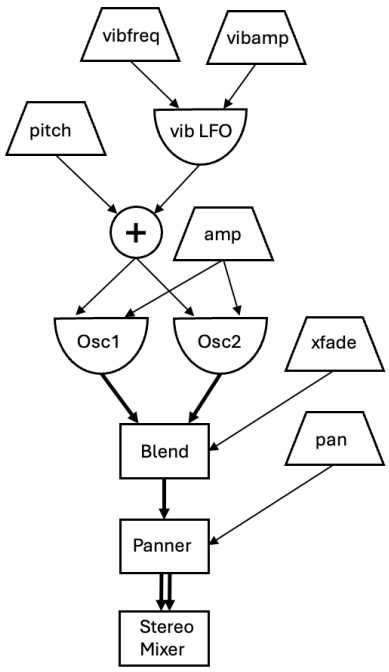

Figure 3. Benchmark for audio processing: 2 cross-faded oscillators (Osc1 and Osc2) with gain, pitch, and vibrato, panned to stereo and added to a stereo mix. Trapezoids are piece-wise-linear envelope generators. Thin arrows represent block-rate signals.

The first study evaluates the impact of block-rate computations. When we replace the block-rate unit generators (mostly envelopes) with audio-rate generators, the computation time increases by a factor of 3.7 to 3.629 s. Block-rate control signals significantly reduce computation time!

The second study considers the cost of supporting multiple channels when only single-channel computation is needed. The benchmark was modified to eliminate the loop that computes each output channel except in the Panner and Stereo Mixer. Per-channel state information, previously requiring an array, was also simplified by assuming one channel. These changes do not affect the computed sound in this benchmark but eliminate the overhead of looping. This optimization decreased computation time by ~~14% to 0.843 s.~~ Thus, the flexibility of multi-channel signals comes at some cost. Note that if there really were multiple channels, single channel unit generators would have to be replicated and invoked separately, offsetting at least some of the savings.

The third study considers the cost of signal termination described in Section 3.4. Since the termination feature is not used in this benchmark, the termination code can simply be removed. The speedup is about 0.8%. This feature is very low in cost, partly because termination tests occur at the block rate.

The fourth study measures the cost of polymorphic inputs. If we know whether inputs are audio-rate or block-rate, we can avoid making an indirect function call to specialized code (as described in Section 3.2) and simply inline the code within the unit generator's compute-block method. This optimization saves only 0.9% of the overall computation time, mainly because polymorphism costs only one indirect method invocation at the block rate.

The fifth study measures the cost of tracing the unit generator graph connections to determine the order of unit generator evaluation as described in Section 3.5. The benchmark was modified to call the unit generators in the proper sequence rather than relying on a recursive graph traversal algorithm. The speedup is about 2.3%, suggesting that trying to pre-sort unit generators into an evaluation order has no significant benefit.

Finally, we look at block sizes. We expected larger block sizes would reduce the overhead by reducing the number of times unit generators are invoked to compute samples. To test this, we varied the block size using *all audio-rate* unit generators and holding constant the total number of samples computed. As shown in Figure 4, run time decreases by over 40% moving from a block size of 2 to 8 (as expected), but larger block sizes run *slower*, and 8 samples appears to be optimal. This is unexpected because larger block sizes require fewer total instructions. Cache behavior is an unlikely explanation because even the L1 cache is much larger than the data. Experiments suggest that simple unit generators compute faster than the results can be written, but small blocks can be absorbed by CPU write buffers. Writes can then complete as the next unit generator is invoked. Thus, (in this benchmark) smaller blocks allow for more parallelism within the internal CPU pipelines.

On the other hand, when *block-rate* unit generators are present (and we saw in the first study that they have a large impact on run time), smaller block sizes raise the control

rate and the cost of computing control signals. With a mix of control rate and audio rate, 16-sample blocks are about 25% slower than 32-sample blocks, and 64-sample blocks are about 8% faster.

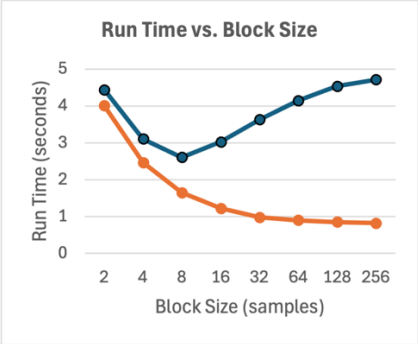

Figure 4. Run time for all audio computation (upper curve) and mixed audio-rate and block-rate computation (lower curve) for different block sizes. The total number of samples computed decreases with block size in the lower curve because the block rate is *sample-rate* / *block-size* and 8 of 13 unit generators run at the block rate. In all cases, using block rate for control decreases the run time relative to all-audio-rate processing.

Yet another way to think about block-based computation is to separate the computation into two parts: the unit-generator setup is the computation needed invoke the unit generator, gather addresses of input signals, and load parameters associated with the computation. The per-sample computation time is the time to compute each sample once the setup is completed. The total time is thus $a + nb$, where a is the setup time, n is the block size, and b is the per-sample computation time. Using our run times for $n = 32$ and $n = 64$ and noting that unit generators run twice as many times in the $n = 32$ case, we get $2(a + 32b) = 0.980$ and $a + 64b = 0.903$. This gives $a = 0.077$ and $b = 0.013$. The ratio $a/b \approx 6$, so the cost of invoking a unit generator is about the same as computing 6 samples. This number depends directly on the amount of work done per sample within the average unit generator, but it explains how larger block sizes deliver diminishing returns when blocks are much larger than 6 samples.

Another consideration in setting the block size is that signals start and stop on block boundaries, so large blocks lead to lower time resolution for starting sound events (unless extra work is done to achieve sub-block timing accuracy). The default 32-sample block size provides sub-millisecond event timing for audio rates of 44,100 or higher.

A final design question is the impact of using configurable unit generators as opposed to compiling all signal processing into a monolithic block of code, as seen in Faust. The inner loops of the benchmark program were reorganized into a block-rate part and an audio-rate part so that a block computation looks roughly like:

```
compute_block() {
    block_rate_part
    for (int i = 0; i < 32; i++) {
        audio_rate_part } }
```

The audio-rate part combines the inner loop bodies of all 5 audio-rate unit-generators so that each pass through the code evaluates the graph of unit generators and produces one stereo frame of output. Only the final output from the Stereo Mixer (see Figure 3) is written to memory. Intermediate unit generator outputs are passed through local variables that are potentially stored in registers, which is faster than reading and writing memory. This approach decreases the execution time by 33%, but it sacrifices the ability to reconfigure the computation in real time, and any changes must be recompiled and reloaded.

Furthermore, it should be noted that if this level of optimization is needed, one can achieve it with a custom unit generator. In fact, we will see in Section 6 how FAUST code, which compiles into this style of monolithic implementation, can be translated directly into Arco unit generators.

5. CONTROLLING ARCO

One of the challenges of the server approach is to design an easy-to-use API given that all communication is through asynchronous messages. As an example of the style of control, one can create a sine oscillator using the message:

```
o2_send_cmd("/arco/sine/new", 0, "iiii",
            id, 1, freq, amp);
```

where `iiii` is an OSC-like type string (four integers), `id` is the identifier for the new unit generator, `1` is the channel count, `freq` is the frequency control, and `amp` is the amplitude control (both identifiers for other unit generators). To update a Const unit generator to a float `value`, one can send:

```
o2_send_cmd("/arco/const/set", 0, "iif",
            cid, 0, value);
```

where `cid` is the identifier for the Const unit generator and `0` is the channel to set.

Unit generators are located using a simple array that maps from small integer identifier to address. To dispose of a unit generator, the application sends an `/arco/free` message. This frees the array slot for reuse, but keeping in mind that unit generators form a directed flow graph, the "freed" unit generator may still be referenced by other unit generators. To avoid "dangling pointers," we use reference counting and only free unit generators when all references are freed.

5.1 Timing

Arco is designed for synchronous real-time audio processing. While audio is running, the logical time of Arco audio computation is synchronous with the audio sample count, and since audio is computed in blocks, time advances by block-period steps.

O2 derives its clock from Arco's logical time, and O2's built-in clock synchronization lets Arco clients share this time reference. By sending commands slightly ahead of real-time and by using timestamped messages (all O2 messages have timestamps), Arco can achieve accurate timing. Messages are delivered at a "clean point" between audio block computations, so most control actions are quantized to block periods (32/44100 or 726 µs by default), which is equivalent to saying "sample accurate" at the control rate. Various special cases such as envelope breakpoints, grains in granular synthesis and even sub-sample accuracy for PSOLA [15], can be achieved within unit generators.

When using the Arco scripting language, scheduler objects can be used to invoke functions or methods at logical

times in terms of seconds or beats, so all computation runs with precise timing. Accurate times are communicated to the Arco server through O2 timestamps.

5.2 The Client-Side Library

For simple applications, the client can send messages to initialize a graph of unit generators and send more messages to control it. However, since all interactions go through messages, the application cannot make a query such as "what audio devices are available now?" or "what is the current value of the gain parameter?" unless it is willing to wait for a reply. Keeping track from the client side of all unit generators and their connections can be challenging. To simplify things for the programmer, we use an object-oriented approach in which every actual unit generator has a client-side "shadow" object that duplicates the state of the Arco unit generator. To create a unit generator, the programmer creates a "shadow object" locally, which, as a side effect, allocates an ID and sends messages to the server to create the real unit generator. With shadow objects, we can query and update interconnections, parameters, and signal types, and when shadow objects are garbage collected, they can automatically free the corresponding unit generator in the audio thread. We have implemented an object library of this sort in Serpent for Arco, and we are porting this to Python.

Our client-side library provides a much higher-level interface to control Arco than sending messages directly. For example, the following creates a tone with multiple harmonics using additive synthesis and some randomization. Note the use of termination (Section 3.4) to free the tone when the envelope ends.

```
// freq, dur, and nharm are parameters
env = pwlb(dur * 0.2, 0.01, dur * 0.8) // envelope
env.term()   // allow envelope to terminate
mix = mix(1).term() // mixer can terminate too
for i = 0 to nharm:     // nharm harmonics
    mix.ins(mix_name(i), // insert input to mixer
            sine(freq * i, 0.8 ** i), env)
```

Each of the constructor functions `pwlb`, `mix`, and `sine` creates a local shadow object for a corresponding unit generator and indirectly sends messages to Arco to create and interconnect actual unit generators.

5.3 Instrument and Synth Abstractions

In practice, synthesis can be more complex than instantiating a few unit generators, raising the question of how larger sub-graphs of unit generators should be created or allocated to create sounds. A realistic example sound is the "Supersaw" sound, which mixes 16 slightly detuned and panned sawtooth waveforms, creating a rich and animated spectrum. Initially, Supersaw was implemented using the "instance" model [16] – creating unit generators as needed for each sound. A 4-note chord of Supersaw tones uses 536 unit generators and takes about 8 ms of computation to create. This is only 15 µs per unit generator (running in the Serpent scripting language – probably the run time in the Arco server, written in C++, is much faster), but the 8 ms "arpeggio" is audible. For this reason, we have developed Synth and Instrument classes that manage sub-graphs of unit generators, allowing for their reuse. To play a sound, a Synth object is presented with a property list of parameter values for the desired sound. The Synth allocates an Instrument from a pool of free instances, updates any parameters that have changed, and starts the Instrument instance. This is more efficient, but more difficult to implement than simply instantiating a new set of unit generators for every sound. Our Arco library allows either approach to be employed.

5.4 Execution Time for Graph Creation

One concern, whether building a new graph for each new sound or reusing an existing graph, is how long does it take? We can expect many actions to be simultaneously scheduled, causing audio computation to stop while unit generators are created or reconfigured. From our Supersaw example, where the creation of shadow objects and messages in the client library took about 8 ms for 4 rather large graphs, we can assume that decoding the messages and building actual unit generators in Arco involves a similar amount of computation. However, Arco is compiled and runs about 2 orders of magnitude faster than the interpreted scripting language in the client. Therefore, we can estimate the processing time in Arco is about 80 µs, or less than 4 sample times. Since audio systems typically buffer far more than 4 samples, the message processing time is negligible. Of course, this is a rough estimate and some well-defined benchmark tests could give a better idea of actual performance.

6. DESIGNING WITH FAUST

FAUST is a high-level functional programming language for specifying DSP algorithms. It has an extensive library of audio filters, generators and effects, and it can compile these algorithms to efficient C++. One of the goals of Arco was to avoid creating yet another library of unit generators, so it was decided to leverage FAUST as an option for creating new unit generators in Arco. In practice, Arco also has unit generators written by hand for spectral analysis, onset detection, granular synthesis, phase vocoder, and other functions that cannot be easily specified in FAUST or that were already implemented in C or C++.

FAUST compiles algorithms to a single block of code that processes the next audio sample and is typically embedded in a loop to compute an entire block of samples, similar to unit generators in other systems. Unlike Arco unit generators, which handle multiple channels and mixed sample rates, one must declare FAUST parameters to be either audio samples or control parameters, and channel counts are fixed. Thus, there is a mismatch between the polymorphic Arco unit generator model and the more static DSP functions compiled by FAUST.

To integrate FAUST code into Arco, we wrote a translator that transforms annotated FAUST code into a set of FAUST programs, specialized for each combination of block-rate and audio-rate inputs. For example, below we give a complete specification for a minimal unit generator that multiplies two signals. The first two lines are non-FAUST annotations. The first says we want an audio-rate

unit generator where inputs can be either audio-rate or block-rate (`ab`). The second line requests a block-rate output with only block-rate (`b`) inputs. Also notice the `interpolated` and `terminate` declarations to direct some Arco-specific translation:

```
mult(x1: ab, x2: ab): a
multb(x1: b, x2: b): b

FAUST

declare name "mult";
declare description "Mult(iply) UGen";
declare interpolated "x1 x2";
declare terminate "x1 x2";
import("stdfaust.lib");
process(x1, x2) = x1 * x2;
```

A preprocessor expands this into multiple FAUST programs, one for each combination of input types, and the FAUST compiler is used to compile each file. Next, a translator program reads and deconstructs the multiple FAUST-generated C++ programs, extracts their computational cores and reconstructs them into a single polymorphic Arco unit generator. In this case, two unit generators are created: `mult` (audio) and `multb` (block-rate).

Arco can also be extended directly by writing unit generators in C++. All unit generators subclass the abstract class `Ugen` and override certain methods for instantiation, deletion, patching, processing samples and handling O2 messages, although some automatic code generation exists to build the O2 message interface. Many examples serve as a guide to implementers. User-written unit generators can be arbitrarily complex. Examples we have implemented include a granular synthesis system, phase vocoder, and a port of Music Onset Detection and Library (Modal) [17].

7. APPLICATIONS

Arco is under active development, but it has been used to re-implement some interactive works for computer and acoustic instruments. (See Figure 5 for a representative graphical interface.) Since O2 is at the foundation of Arco, we have investigated ways to connect Arco to other processes and devices. In one application, we use a computer-vision-based hand-tracker that runs in a browser to communicate over web sockets to Arco, creating an interesting controller for sound (see Figure 6). In this application, position controls pitch, width controls filter cutoff, and other gestures adjust the gain of a simple instrument. Filters in Arco are applied to smooth the low-frame-rate control information. In another application, we use O2 to send audio between Arco and a process running RAVE [18] using PyTorch. These are just two examples of integrating Arco with machine learning systems. O2 forms the "glue" for inter-process communication. Arco has built-in audio analysis unit generators for pitch, amplitude, spectral features, chord recognition and onset detection, which could be interesting inputs for an interactive machine-learning system. Soon, we expect to have a Python library for controlling Arco, where Arco will run as a server process. We also hope to encourage the use of Arco as a lightweight and highly configurable real-time signal processor for microcontrollers.

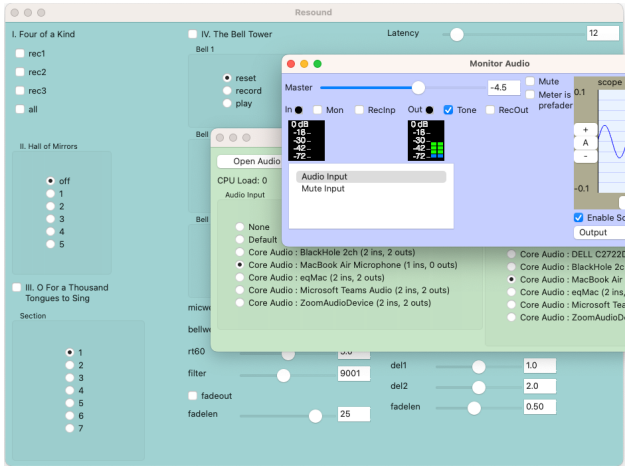

Figure 5. An Arco implementation of "Resound" for trumpet and interactive computer. Control and interface are implemented in the Serpent scripting language. Arco is linked to Serpent as a library.

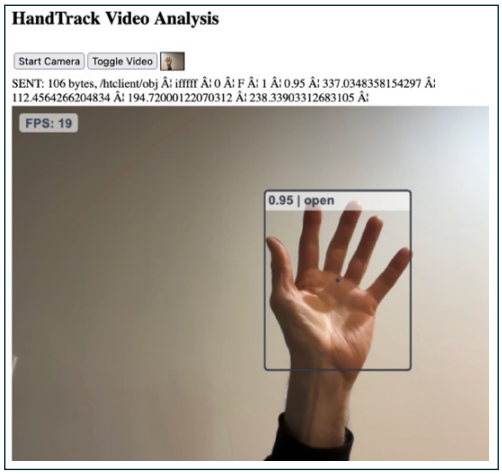

Figure 6. Video control of synthesis application combining Handtrack.js (github.com/victordibia/handtrack.js) and Arco for sound synthesis, using O2 for communication. O2 implements a simple web server, the O2lite protocol running over web sockets, and an O2lite client in Javascript, so minimal new code is needed for these programs to interoperate.

8. SUMMARY AND CONCLUSIONS

Arco is a software music signal processing server and library intended to be flexible, extensible, and to enable modular construction of music systems as opposed to trying to combine many functions into a single monolithic package. Arco aims to do one thing well, which is to process audio in real time by configuring unit generators. Arco uses O2 messages to communicate with other processes that might offer interactive control, graphical user interfaces, machine learning and inferencing, etc.

The design of Arco has been described, including details of its unit generators. The unit generator model in any system is the result of many design decisions. We have measured the impact of multiple design choices including block-oriented processing, multi-channel unit generators, supporting the notion of signal termination to allow

automatic cleanup of unwanted unit generators, polymorphic inputs (audio rate and block rate), dynamic ordering of unit generator execution, and block size. Of these design decisions, supporting control-rate processing with linear interpolation for up-sampling within audio-rate processors is a clear winner offering significant speedup. Other features such as multi-channel unit generators, polymorphic inputs, and termination-based cleanup have only small impacts on performance and seem worth including.

One surprise in our benchmarking study was the U-shaped curve plotting run-time vs. block size. It was expected that larger blocks would be more efficient, but at least for our benchmark, the optimal block size was only 8 samples, *but only for audio-rate computation*. Larger block sizes are best when there is a mix of audio-rate and block-rate unit generators, but larger blocks yield rapidly diminishing returns. The main factor influencing block size should be using a high enough control rate that it can be widely used, supporting the default size of 32 in Arco.

New unit generators in Arco can be created using FAUST. Thus, we can combine the excellent DSP library resources of FAUST with the flexibility and run-time configuration offered by Arco.

Arco is free, open source, and available (github.com/rbdannenberg/arco). Interested developers and users are encouraged to contact the author.

ACKNOWLEDGMENTS

Contributors to Arco include Yudong Chen, Joseph Hsu, Vivek Mohan, Chukwuemeka Nkama, and Mark Zhou.

REFERENCES

[1] V. Lazzarini, S. Yi, J. Ffitch, J. Heintz, Ø. Brandtsegg, and I. McCurdy. *Csound: A Sound and Music Computing System*. Springer, 2016.

[2] R. Dannenberg, "The Implementation of Nyquist, a Sound Synthesis Language," *Computer Music Journal*, vol. 21, no. 3, pp. 71-82, 1997.

[3] J. McCartney, "Rethinking the Computer Music Language: Supercollider," *Computer Music J*ournal, vol. 26, no. 4, pp. 61–68, 2002.

[4] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, 2015.

[5] M. Puckett, "Pure Data," *Proceedings, International Computer Music Conference*, Hong Kong, 1996, pp. 224–227.

[6] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[7] P. R. Cook and G. Scavone, "The Synthesis ToolKit (STK)," *Proceedings of the 1999 International Computer Music Conference*, Beijing, pp. 164-166.

[8] J. Storer, "JUCE: Jules' Utility Class Extensions" [computer software], Raw Material Software Ltd., 2024.

[9] J. de Guzman, "Q Audio DSP Library" [computer software], Cycfi Research, 2024. (available at https://github.com/cycfi/q).

[10] X. Amatriain, P. Arumi, and D. Garcia, "CLAM: a Framework for Efficient and Rapid Development of Cross-Platform Audio Applications," *Proceedings of ACM Multimedia*, 2006, pp. 951–954.

[11] Y. Orlarey, D. Fober, and S. Letz, "FAUST: An Efficient Functional Approach to DSP Programming," *New Computational Paradigms for Computer Music*, Editions Delatour, 2009.

[12] R. Dannenberg, "A Language for Interactive Audio Applications," *Proceedings of the 2002 International Computer Music Conference*, Gothenburg, 2002, pp. 509-15.

[13] R. Dannenberg, "Communication for Real-Time Music Systems: An Overview of O2," *Computer Music Journal*, vol. 45, no. 4, pp. 7-19, 2022.

[14] R. Dannenberg and N. Thompson, "Real-Time Software Synthesis on Superscalar Architectures," *Computer Music Journal*, vol. 21, no. 3, pp. 83-94, 1997.

[15] P. Dutilleux, G. De Poli, A. von dem Knesebeck, and U. Zölzer, "Time-segment processing (chapter 6)," *DAFX: Digital Audio Effects, Second Edition*, U. Zölzer, ed., pp. 185–217, 2011.

[16] R. Dannenberg, D. Rubine, and T. Neuendorffer, "The Resource-Instance Model of Music Representation," *Proceedings of the 1991 International Computer Music Conference*, Montreal, 1991, pp. 428-432.

[17] J. Gover, *Sinusoids, Noise and Transients: Spectral Analysis, Feature Detection and Real-Time Transformations of Audio Signals for Musical Applications*, PhD thesis, National University of Ireland Maynooth, 2012.

[18] A. Caillon and P. Esling, "RAVE: A variational autoencoder for fast and high-quality neural audio synthesis," arXiv:2111.05011, 2021.