Flexible Scheduling for DataFlow Audio Processing

Neil Burroughs, Adam Parkin, and George Tzanetakis
Department of Computer Science, University of Victoria
{inb, aparkin, gtzan}@cs.uvic.ca

Abstract

The notions of audio and control rate have been a pervasive feature of audio programming languages and environments. Real-time computer music systems depend on schedulers to coordinate and order the execution of many tasks over the course of time. In this paper we describe the scheduling infrastructure of Marsyas-0.2, an open source framework for audio analysis and synthesis. We describe how to support multiple, simultaneous, dynamic control rates while retaining the efficiency of block audio processing. In addition we show how timers and events can be abstracted and decoupled from the scheduler in an extensible way. Specific types of supported events such as control updates, implicit patching, wires and expressions are described. In addition, we show how multiple timers based on sample-time, real-time and virtual-time can be utilized. The work in this paper has been motivated by the precise handling of time in the Chuck audio programming language and therefore we show how a simple Chuck program can be expressed in the Marsyas Scripting Language (MSL).

1 Introduction

There is a large ecology of programming languages, frameworks, and environments for the analysis and synthesis of audio and music signals. The design of such computer music systems is especially challenging as there are many potentially conflicting requirements that need to be addressed. Three of the most important requirements are: 1) efficient audio processing 2) interactivity, and 3) expressivity. Even today, with laptops that are as powerful as supercomputers were five years ago, practitioners of computer music are frequently constrained by computational efficiency (for example by limits in the number of simultaneous oscillators that are supported). In addition to efficient computation, computer music systems must be able to respond quickly to external events such as user interface actions, sensor measurements or precomposed actions. Finally computer music programming languages and environments should provide a small number of high level concepts, constructs and building blocks that can be combined to accomplish a large variety of tasks.

The central focus of this paper is the handling of time in *Marsyas* an open source framework for audio analysis, retrieval and synthesis [1]. The concepts of audio and control rate are pervasive in many computer music systems including the Max/PD family (Puckette 2002), SuperCollider (Mccartney 2002), and CSound (Boulanger 2000). More recently, handling multiple concurrent and adjustable control rates has been demonstrated in *Chuck* (Wang and Cook 2004).

Marsyas is based on a dataflow model of computation which supports easy dynamic adjusting of audio rate for arbitrarily large networks of audio processing units. We describe the new scheduling/timing infrastructure developed in *Marsyas* to support multiple control rates. This infrastructure also supports multiple user-defined Timers and Events by abstracting these notions from the Scheduler. Our work has been motivated by ideas in *Chuck*, therefore we show how a simple *Chuck* program can be expressed in the *Marsyas Scripting Language* (MSL) without losing the computational efficiency of block processing.

2 Related Work

All real-time computer music systems depend upon schedulers as a means of coordinating and ordering the execution of many small tasks over the course of time. A good overview of real-time scheduling in Computer Music is provided in (Dannenberg 1989). *Marsyas* started as a framework for audio analysis with specific emphasis to Music Information Retrieval (MIR) applications. The 0.2 version (a complete redesign/rewrite) was initiated in order to provide synthesis functionality similar to the *Synthesis Toolkit* (STK) (Cook and Scavone 1999). Marsyas is based on synchronous dataflow block processing with block-synchronous control updates for efficient audio processing. Implicit patching (Bray and Tzanetakis 2005) doesn't enforce a fixed processing block size across the dataflow network and enables dynamic adjustment of buffer sizes (audio rate) at run time.

[1] `http://marsyas.sourceforge.net`

The development of Marsyas has been influenced by a variety of existing systems and ideas. A detailed description of an object-oriented metamodel for audio processing with clear separation of synchronous dataflow and asynchronous control flow is provided in (Xavier 2005). The default path-based naming of controls utilized in Marsyas is inspired by Open Sound Control (OSC) (Wright and Freed 1997), and the explicit representation of time in the processing of slices is inspired by SDIF (Schwarz and Wright 2000). The use of implicit patching using composites is similar to the block algebra in *Faust* (Graef, Kersten, and Orlarey 2006). Another influence has been the use of multiple control rates in Aura, a flexible object-oriented software synthesis system (Dannenberg and Brandt 1996). The timing infrastructure described in this paper was motivated by ideas from *Chuck* (Wang and Cook 2004). *Chuck* also influenced syntactic decisions for the Marsyas Scripting Language (MSL).

3 The Marsyas Dataflow Architecture

Marsyas-0.2 is an open source software framework written in C++. A variety of existing building blocks are provided as dataflow components that can be composed to form more complicated algorithms. In addition, it is straightforward to extend the framework with new building blocks. In this section we provide a quick overview of the architecture in order to provide context for the remainder of the paper. *MarSystems* are the processing nodes of the dataflow network. Complicated audio processing networks can be expressed as a single large "Composite" *MarSystem* which is assembled by implicitly patching basic *MarSystems* (Bray and Tzanetakis 2005). Figure 1 shows how a Series composite consisting of a SoundFileSource src, Gain g, and AudioSink dest can be assembled in C++. At every iteration of the loop the audio rate is incremented starting from 1 sample (similar to *Chuck* or *STK*) until the block size of 1000 samples is reached. All the intermediate shared buffers between the *MarSystems* are adjusted automatically and the sound plays without interruption. Although this example might seem artificial dynamically adjusting window size is useful for analysis algorithms such as pitch synchronous overlap-add (PSOLA).

MarSystems process chunks of data called *Slices*. *Slices* can be viewed as a generalization of blocks of samples and are matrices of floating point numbers characterized by three parameters: number of samples (things that "occur" at different times), number of observations (things that "occur" at the same time) and sampling rate. Figure 2 shows a *MarSystem* for spectral processing that converts an incoming audio buffer of 512 samples of 1 observation at a sampling rate of 22050 to 1 samples of 512 observations (the FFT bins) at the lower sampling rate of 22050/512. Controls can be used to modify the behavior of *MarSystems* and utilize path notation similar to OSC(e.g. /net/gain/frequency). Updating controls is done in a slice synchronous manner.

4 Scheduling Infrastructure

Scheduling is central to any computer music system. A scheduling request consists of an event and a time. The scheduler keeps track of pending requests. Computer music schedulers use times which are typically references to a single clock that is presumed to correspond to real (physical) time. In systems with explicit handling of time, like *Chuck*, events can be scheduled in sample time. Sample time is defined directly in terms of the number of audio samples generated. *Chuck* processes individual samples and therefore provides sample accurate timing. *Marsyas* can do sample accurate timing by setting inSamples to 1 but also allows to tradeoff more efficient block processing at the cost of more coarse timing at block boundaries. If audio computation takes longer than real-time then system time and sample time can be different. Both of these notions of time can be useful. In computer music programs it is also often convenient to have a time reference or references that do not correspond to real time. For example consider scheduling events in beats that are defined in relation to a conductor's baton. In this section we show how multiple notions of time and events are supported through object-oriented abstractions in *Marsyas*.

Each *MarSystem* object has its own *Virtual Scheduler* that manages an arbitrary number of *Event Schedulers*. Each *Event Scheduler* contains its own *Timer* that controls the rate at which time passes and events are dispatched. *Schedulers* themselves do not keep track of time but leave this task solely to the *Timer*. Structured this way events may be scheduled to any number of different *Timers*.

A *Timer* in *Marsyas* is any object that can read from a time source and trigger some action on each tick. A timer must also provide some way to specify units of time in its time base. The one restriction in defining a time source is that time must always advance. *Timers* are definable by the user pro-

```
MarSystem* net = mng.create("Series", "net");
net->addMarSystem(src);
net->addMarSys(g);
net->addMarSys(dest);

for (int i=1; i<1000; i++) {
  net->updctrl("natural/inSamples", i);
  net->tick(); }
```

Figure 1: Marsyas C++ dynamic audio rate adjustment

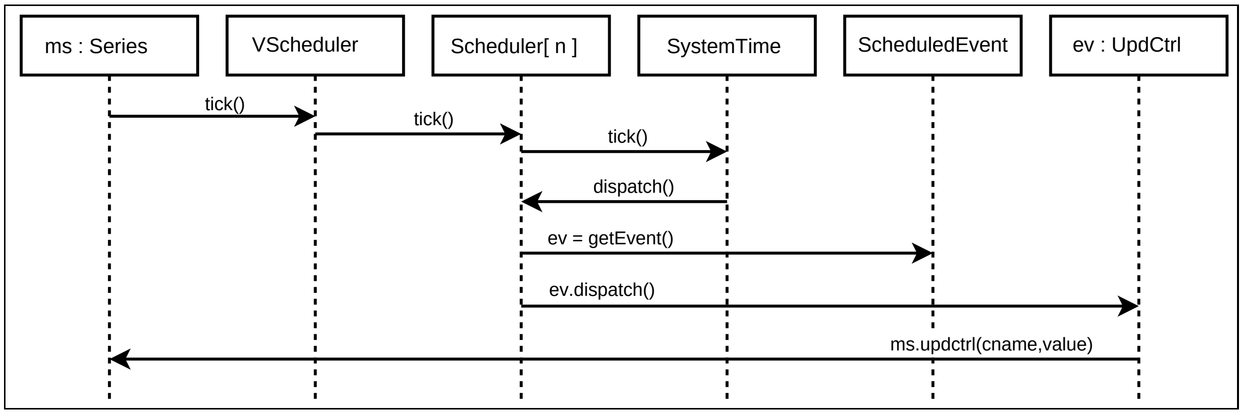

Figure 3: UML sequence diagram of event dispatch

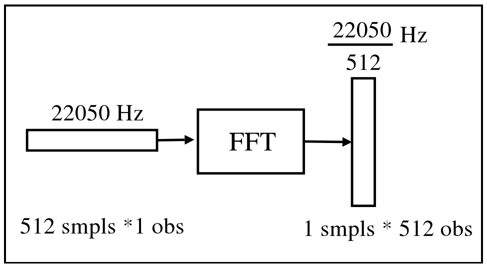

Figure 2: MarSystem and corresponding slices for spectral processing

vided they support the *AbstractTimer* interface. The interface requires specification of the following: a method for determining the interval of time since the last reading, a method for comparing time intervals for that particular *Timer*, a trigger method which calls the Scheduler dispatch, and a method for converting time representations to the specific notion of time used by the *Timer*. This generalization of *Timers* allows for many different possibilities in controlling event scheduling. Linear and non-linear advancing are both possible.

Events are user definable actions that are "frozen" until dispatched. They are distinct from the normal flow of audio data. There are no restrictions on the types of events that can be defined. Perhaps the most common event is updating a *MarSystem* control. Another example is the *Wire* event which updates the value of a control based on the value of another control (similar to a control wire in Max/MSP, PD). Expressions involving multiple controls, constants and user defined functions are also *Events*. *Events* must supply a dispatch() method that the scheduler can call at the appropriate time. *Events* may take place immediately or some time in the future. The time at which an *event* happens may be specified by the user and depends on the *timer* that the requested time is with respect to. An *Event* is then sent to the scheduler and removed when its time interval has passed. The *Scheduler* then calls the *dispatch()* method on the event.

Figure 4 shows a UML class diagram of the scheduling architecture. Notice how multiple, user-defined *Timers* and *Events* can be supported by abstraction and are decoupled from the *Scheduler*. The UML sequence diagram of figure 3 shows the order of method calls between objects for performing a control update (a specific type of event).

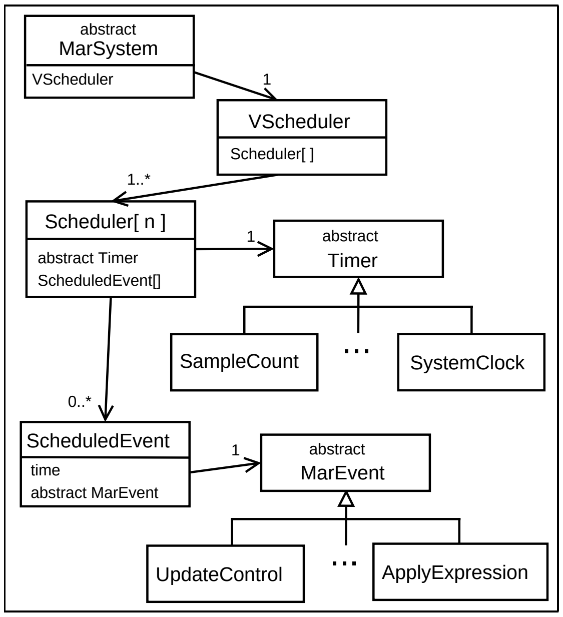

Figure 4: UML class diagram of the scheduler's architecture.

```
sinosc s => dac;
440.0 => s.freq;
while ( true ) {
  (Math.rand() * 10000.0) => s.sfreq;
  100::ms => now; }
```

Figure 5: ChucK example

```
Series net is [ SineSource s,
                AudioSink dac]

do [ 440.0 => net/s/frequency ]
do [ (Math.rand() * 10000)
        => net/s/frequency
] every 100ms using SampleTimer
run net run
```

Figure 6: MSL example

5 Marsyas Scripting Language

MSL is a scripting language for building and controlling *MarSystem* networks at runtime. Lexical analysis (or scanning) is performed using the Flex scanner generator, and parsing performed using the Bison parser generator. The output of the parsing stage from Bison is an abstract syntax tree representation of the MSL script, which is then traversed to generate and execute the equivalent *Marsyas* C++ code.

Figures 5 and 6 show how a simple example in *Chuck* can be written in *MSL*. Every 100 milliseconds the sine oscillator is set to a random frequency between 0-10kHz. The “do” construct allows multiple events to be scheduled at particular times based on a Timer. Figure 7 shows a more complicated example with two voice polyphony using a Fanout composite. Two sine oscillators are controlled by separate timers one based on SystemTime and the other based on ConductorTime which is based on MIDI input. Note that the block size determined by inSamples is completely decoupled from the timers and can be changed by updating a control. It is possible to instantiate in *MSL* the entire feature extraction method described in (Tzanetakis and Cook 2002).

6 Future Work

The scheduling system has been designed to work on a single host. We are working on making a distributed version of the scheduler that allows events to be posted across a network of hosts. An effort is also underway to add more event types to Marsyas and to see to what extent the behaviour of the system can be modified using events.

```
Series net is [
    Fanout mix is [ SineSource src1,
                    SineSource src2 ],
    Sum    sum,
    AudioSink dac
]
do [ (Math.rand() * 10000) + 100
        => net/mix/src1/frequency
   ] every 2beats using ConductorTimer
do [ (net/mix/src2/frequency + 400) % 10000
        => net/mix/src2/frequency
   ] every 0.5s using SystemTimer
run net run
```

Figure 7: MSL example with multiple timer

References

Boulanger, R. (2000). *The Csound book*. MIT Press.

Bray, S. and G. Tzanetakis (2005). Implicit patching for dataflow-based audio analysis and synthesis. In *Proc. Int. Computer Music Conf. (ICMC)*.

Cook, P. and G. Scavone (1999, October). The Synthesis Toolkit (STK), version 2.1. In *Proc. Int. Computer Music Conf. ICMC*, Beijing, China. ICMA.

Dannenberg, R. (1989). Real-time scheduling and computer accompaniment. In M.Mathews and J.R.Pierce (Eds.), *Current Directions in Computer Music Research*, pp. 225–261. Cambridge, MA: MIT Press.

Dannenberg, R. and E. Brandt (1996). A flexible real-time software synthesis system. In *Proc. Int. Computer Music Conf. (ICMC)*, pp. 270–273.

Graef, A., S. Kersten, and Y. Orlarey (2006). Dsp programming with faust, q and supercollider. In LAC (Ed.), *Linux Audio Conference 2006*.

Mccartney, J. (2002). Rethinking the computer music language: Supercollider. *Computer Music Journal 26*(4), 61–68.

Puckette, M. (2002). Max at seventeen. In *Computer Music Journal*, Volume 26, No. 4, pp. 31–43.

Schwarz, D. and M. Wright (2000). Extensions and applications of the sdif sound description interchange format. In *Proc. Int. Computer Music Conf.*

Tzanetakis, G. and P. Cook (2002, July). Musical Genre Classification of Audio Signals. *IEEE Trans. on Speech and Audio Processing 10*(5).

Wang, G. and P. Cook (2004). Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proc. ACM Int. Conf. on Multimedia*, pp. 812–815.

Wright, M. and A. Freed (1997). Open sound control: A new protocol for communicating with sound syntesizers. In *Proc. Int. Computer Music Conf. (ICMC)*, Thessaloniki, Greece.

Xavier, A. (2005). *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. Ph. D. thesis, Univ. of Pompeu Fabra.