Exploiting Parallelism in Real-Time Music and Audio Applications

Amar Chaudhary and Adrian Freed

Center For New Music and Audio Technologies
University of California, Berkeley, CA 94709, USA
{amar,adrian}@cnmat.berkeley.edu

Abstract. We introduce a scalable, extensible object-oriented system developed primarily for signal processing and synthesis for musical and multimedia applications. The primary performance issue with these applications concerns functions of discrete-time. We describe novel techniques to exploit fine-grain parallelism in the calculation of these functions that allow users to express them at a high-level in C++. The second half of the paper describes the scheduling strategies used to exploit symmetric multiprocessors with emphasis on special hard real-time constraints required. This scheduler efficiently assigns processors to multiple synchronous dataflow graphs triggered from independent hardware clocks and satisfies constraints on input/output latency, and execution order.

1 Introduction

OSW, or "Open Sound World" is a scalable, extensible object-oriented language that allows sound designers and musicians to process sound in response to expressive real-time control. OSW is a "dataflow programming language," similar to Ptolemy [1] and Max/MSP [2] in the signal-processing and computer-music communities, respectively. In the dataflow programming model of OSW, primitive components, called *transforms*, are connected together to form dataflow networks, called *patches*. Each component accepts data from incoming connections and sends processed results via outgoing connections. OSW is also an "object-oriented" language, in that components are *instances* of *classes* that specify their structure and behavior. OSW allows users to develop at multiple levels including visual patching (as illustrated in figure 1), high-level C++ and scripting.

OSW includes a large set of standard transforms for basic event and signal processing. A current list of the standard transforms included in OSW can be found at `http://www.cnmat.berkeley.edu/OSW`. The set of available transforms can be easily extended to include more advanced operations. Since the data types used by transforms are C++ types (i.e., classes or primitive scalars), it is relatively straightforward to add new data types as well.

2 Exploiting Fine-Grain Parallelism in OSW

The recently completed standardization effort for C++ was not a formal codification of existing practice with the language. Many years of the effort involved

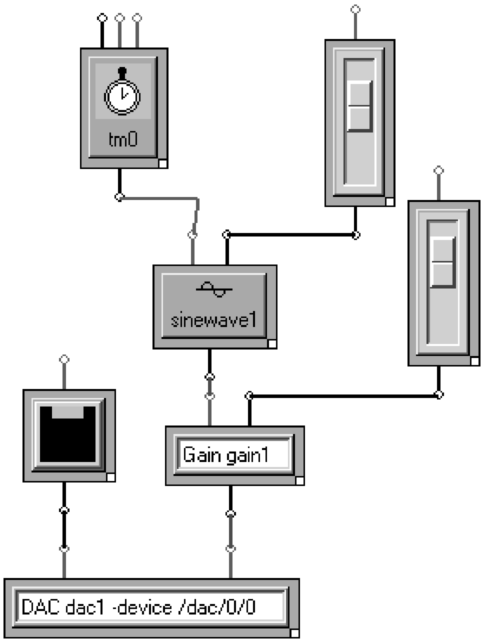

Fig. 1. A patch that plays a pure tone with varying frequency and amplitude

the introduction of entirely new features many of which directly address efficiency issues that have prevented C++ from use in reactive music software. These features allow high-level specifications to be converted to efficient code by optimizing compilers. Although many of these techniques are being adopted in the numerical-computing community, their use has not been explored in signal-processing and music synthesis applications. Additional techniques exploit mathematical properties of the functions of time used extensively in these applications.

2.1 The Externalizer

OSW provides a graphical tool called the Externalizer that allows users to "peer under the hood" of a transform and extend its behavior without a deep knowledge of C++ or low-level efficiency concerns.

The Externalizer presents a transform implementation as a collection of inlets, outlets, *state variables* and *activation expressions* that a user can view or modify. A state variable is a public variable of a transform that can be queried or modified by other transforms in OSW. Inlets and outlets are special cases of state variables used in connections. When an outlet is modified, it sets a connected inlet to that value as well.

An activation expression is a piece of C++ code that is executed when certain inlets or state variables are modified. If an activation depends on more than one variable, then all the variables must be modified before it will be executed. An activation can occur immediately, or delayed for a specified amount of time before running. An activation expression is specified by the subset of inlets or state variables that trigger this activation, whether this activation should occur immediately after the variables are changed or be delayed by a certain amount of time, and the code that should be executed. An activation expression can also be assigned an optional integer that determines its relative order among all expressions triggered by a particular variable.

Consider the following specification of Sinewave, a transform that implements a simple sinusoid oscillator:

Sinewave

Generates a pure tone (i.e., sine wave) signal.

	Name	Type	Default
Inlets	timeIn	Time	
	frequency	float	440.0
Outlets	samplesOut	Samples	
State Variables	prevTime	Time	0.0
Inherited	SampleRate	float	44100.0
	NumberOfSamples	int	128

Activation Expression	activation1
Depends on	timeIn
Delay	none

```
samplesOut = sin(TWOPI * frequency
                        * Range(prevTime,timeIn,NumberOfSamples));
prevTime = timeIn;
```

Whenever a new time value is received via timeIn, the sine function is computed with frequency frequency over a range of NumberOfSamples values from prevTime and timeIn[1]. More specifically, the Range function creates a vector of interpolated values between prevTime to timeIn. Computing the sine function over the product of 2π frequency and this vector yields another vector containing the desired output samples, which are then sent to samplesOut. The ending time value (i.e., timeIn) will be the starting time value the next time this expression is evaluated. The state variables NumberOfSamples and SampleRate are *inherited* from a more general class of transforms that share a set of common features. In this case, the more general class is *time-domain transforms* that manipulate time-domain samples. Inherited variables can be used in activation expressions just like other state variables.

[1] In practice, the formula used in this activation expression should be replaced with one that does not cause phase discontinuities when the frequency changes

Users can invoke the Externalizer by selecting a transform to examine from a patch. The specification of the transform class is then displayed. This specification includes the name of the transform class, brief description and references to documentation, any special base classes (e.g., time-domain transform) and lists of its inlets, outlets, state variables, inherited variables and activation expressions. The user can then select any of these items to call up detailed information and make modifications. When the user finishes modifying a transform specification, it is automatically converted to a C++ class that implements the transform. The C++ code is then compiled into a dynamic library which can be loaded into OSW and used to instantiate transforms of the new class.

The Externalizer also allows users to define new data types for transform variables. A new data type consists of a number of named fields, each of which must be a preexisting OSW data type. A conversion to and from the string data type must be included for human-readable input and output. The data type specification is then converted to a C++ `struct` definition for use in transforms.

2.2 The `osw::vector<T>` Class

Externalizer specifications allow users to specify activation expressions using intuitive, familiar mathematical definitions instead of hand-optimized computer code. This is achieved through the use of function and operator overloading [3] in expressions that use OSW's vector template class, `osw::vector<T>`[2]. The `Samples` type is actually a synonym for `osw::vector<float>`, an optimized vector of single-precision floating-point numbers.

When a standard arithmetic operator or math function is called with a vector argument, a *composition closure object* [4] is created that evaluates the entire expression for each element in the vector, eliminating the need for temporary storage in complex arithmetic expressions. Like functors, calls to closure objects are inlined. Consider the activation expression from Sinewave:

```
samplesOut = sin(TWOPI*frequency*Range(prevTime,timeIn,NumberOfSamples));
```

A naive implementation would first create a vector for the range with `NumberOfSamples` elements, a temporary for the multiplication by `TWOPI` and `frequency` and a final vector containing the sine of each element from the temporary. The intermediate allocation wastes time and space, and also breaks standard loop optimizations, such as unrolling and opportunities for instruction-level parallelism. The composition closure object defines a function that creates only the final vector. Each element of the final vector is assigned the sine of the multiplication of `TWOPI`, `Frequency` and the corresponding element of the range. The temporary allocation has been eliminated, and the resulting simple loop can now be further optimized by the compiler to exploit instruction parallelism.

The `osw::vector<T>` class also overrides the standard C++ allocators to implement a deferred reference-counted memory manager. In this scheme, further

[2] The explicit namespace `osw::` is used to avoid confusion with the Standard Template Library class `vector<T>`.

allocations are saved when a vector is simply passed without modification from one variable to another; its reference count is increased whenever it is assigned to a new variable, and decreased when it unassigned or a variable goes out of scope. If the reference count decreases to zero, the vector is deallocated and its memory is returned to a shared pool for use by other vectors.

2.3 Optimizations on Functions of Time

Composition closure, functors and operator overloading are already well-known techniques in the numeric community, and are incorporated into several C++ numeric libraries [3]. We can exploit additional optimizations for functions of discrete time that are used extensively in signal processing applications.

Consider the `Time` data type used in the Sinewave example. `Time` can be implemented as a discrete iterator with regular sampling at a default rate of 44100kHz:

```
class Time {
      double time; const double sampling_interval;
public:
      Time(double srate=44100.0): time(0.0),sampling_interval(1.0/srate) {}
      operator double() { return time; }
      friend double operator *(double f, Time &t){ return f * t.time; }
      Time& operator ++() { time += sampling_interval; return *this; }
};
```

The above implementation has the virtue of simplicity, but is too slow for real-time work since the transcendental sine function is computed for every sample point. The usual way to address this deficiency is to replace calls to the mathematical functions with calls to low-level optimized functions. We prefer another approach where we evolve the `Time` class to include formal descriptions of the mathematical identities behind the optimizations, leaving the compiler to deal with the details of exploiting the identities for the particular machine the code will run on. The first step is to change the time representation to more honestly reflect that we are really computing discrete-time sequences:

```
class Time {
     int sample_count; const double sampling_interval;
public:
     Time(double srate=44100.0):sample_count(0), sampling_interval(1.0/srate) {}
     operator double() { return sample_count*sampling_interval; }
     friend double operator *(double f, Time &t) {
         return f * t.sample_count*t.sampling_interval;
     }
     Time& operator ++() { ++sample_count; return *this; }
};
```

Noting that the sequence $s_0 = 1, \ldots, s_i = s_{i-1}e^k$ computes $e^k i$ for real and complex values of k, we can introduce a new class to optimize computation of sinusoids using the imaginary result of Euler's identity $e^{i\theta} = \cos\theta + i\sin\theta$:

```
class sinstate {
     complex<double> value, factor;
  public:
     double sin(double k, int i) {
       return (i==0) ?
                (factor = ::exp(complex<double>(0.0,k)), value=1.0) :
                imag(value *= factor);
     }
};
```

For the compiler to use these optimization classes, we have to specify when they may substitute for sub-expressions involving sinusoid functions. We first use a composition-closure object (e.g., `defmul` below) to defer multiplication of double constants and time variables. Then we can define a sinusoid function that operate on defmul objects:

```
class Time {
  private:
     mutable int sample_count;
     const double sampling_interval;
     mutable int expindex, sinindex;

     class sinstate {
         mutable complex<double> value, factor;
       public:
         double sin(double k, int i) const {
             return (i==0) ?
                    (factor = ::exp(complex<double>(0.0,k)), value=1.0) :
                    imag(value *= factor);
         }
     };
     vector<sinstate> sinstates;

  public:
    Time (double srate=44100.0, int depth=MAXDEPTH):
         sampling_interval(1.0/srate), sinstates(depth),expstates(depth),
         expindex(0),sinindex(0), sample_count(0) {}
    operator double() const {
        return double(sample_count) *sampling_interval;
    }
    const Time& operator ++() const {
        expindex=sinindex=0; ++sample_count; return *this;
```

```
    }
    struct defmul{ // deferred multiplication
        const Time& t;
        const double& f;
        defmul(const Time& at, const double& af): t(at), f(af){}
        operator double() { return f * double(t); }
    };
    friend defmul operator *(const double f, const Time &t) { return defmul(t,f);}
    friend defmul operator *(const Time &t, const double f) { return defmul(t,f);}

    friend double sin(const defmul &dm) {
        return dm.t.sinstates[dm.t.sinindex++].sin(
            dm.f*dm.t.sampling_interval, dm.t.sample_count);
    }
};
```

This technique can be extended to the other common trigonometric and exponential functions used in signal processing.

3 Coarse-Grain Parallelism

The real-time scheduler used by OSW supports symmetric multiprocessor computers, as well as configurations with multiple audio devices and time sources. Fine-grain synchronization primitives (i.e., locks) are included to protect against non-deterministic behavior that arises in such parallel systems without severely compromising performance.

3.1 A Parallel Scheduler for OSW

OSW patches are a special case of dataflow networks [5]. Music and audio applications exhibit a coarser-grain parallelism in which long chains of transforms must be executed in sequence but multiple such chains can run in parallel.

This notion of coarse-grain parallelism can be formalized. Consider all finite sequences of transforms $X = T_1, T_2, \ldots, T_n$ such that T_1 is connected to T_2, etc. If there exists no transform T such that an outlet of $T_1, \ldots, T_{n-1}$ connects to an inlet of T or an outlet of T connects to an inlet of $T_2, \ldots, T_n$, then X is a *chain*. More informally, a chain is a sequence of connected transforms in which there are no branches. We say that X is a *maximal chain* if there is no chain Y such that $X \subset Y$ (i.e., if we add any more transforms to X, X will no longer be a chain). Each maximal chain must be scheduled sequentially, while separate maximal chains can run in parallel. Thus, maximal chains are considered the formal unit of parallel computation in OSW. Examples of maximal chains that often occur in real programs include processing several channels of audio, or synthesizers with multiple voices (i.e., polyphony).

Given N processors, we instantiate N threads for processing the scheduled chains. Each chain-processing thread executes the following loop *ad infinitum*:

loop
  pop a chain off the queue and execute it.
end loop

The following greedy decision rule is used by a transform after an outlet has just been assigned to determine whether or not its connections are part of the current maximal chain:

if this transform has more than one outlet **then**
  The transform connected to this outlet begins a new maximal chain.
  Add it to the queue and return
else
  The next transform is part of the current chain.
  Assign the outlet value to the connected inlet and evaluate its activation expressions...
end if

As described below, audio devices have activation expressions that both trigger and are triggered by a clock when an output device needs more samples. Thus, these tasks are already scheduled as part of maximal chains. This is how the scheduler integrates real-time latency constraints. However, there remain necessary tasks to perform, including scheduling asynchronous events and deferred deallocation of reference-counted memory. We can insert deferred deallocation in between the execution of chains:

loop
  pop a chain off the queue and execute it.
  Do deferred deallocations if no other thread is doing this.
end loop

Only one thread is allowed to perform deallocations at any given time, thus protecting the common memory pool and also preventing additional processors from being wasted on memory management. Alternatively, deferred deallocation can be performed by a separate thread that runs at lower priority (i.e., is given less processing time) than the chain-processing threads. Additional threads are needed for handling the user interface as well as some asynchronous input devices, such as MIDI input ports[3]. Although this means there will be more threads than processors, the asynchronous input threads will likely be idle much of the time and therefore do not need to be assigned dedicated processors.

3.2 Reactive Real-Time Constraints

OSW is designed for implementing *reactive real-time* audio and music applications. Reactive real-time involves maintaining output quality while minimizing *latency*, the delay between input and output of the system, and *jitter*, the change

[3] The need for separate threads for devices depends on the device and the operating system.

in latency over time [6]. Because of the combination of human sensitivity for jitter and the need for reactive response to gestures, we have set the latency goal for the OSW scheduler of 10 ± 1ms [7] [8].

It is the job of the audio output device to implement these constraints. The audio output device is a transform that has two state variables that represent these constraints. SampleBufferSize is the number of samples that are sent to the device at once, and TargetLatency is the total number of samples that are allowed to be placed in the output queue awaiting realization by the sound hardware. In order to fulfill real-time requirements, the audio output device has to be able to determine when the signal processing that produces the samples it will output is performed. This is accomplished by controlling *virtual time sources* via the *clock*. Clocks measure real time from hardware devices. Virtual time [9] is a scaleable representation of time. In OSW, virtual time is handled by *time machines*, transforms that scale input from clocks or other time machines. The audio output device includes an activation expression that depends on a clock:

```
FlushSamples();
while(SamplesInQueue() > TargetLatency) {
Wait();
}
clock = clock + SampleBufferSize / SampleRate;
```

The `Wait` operation is system dependent, and may include deferring to another thread or process to perform other events such as MIDI input. `FlushSamples` outputs the samples for this period. (The number of samples output is the sample-buffer size.)

When the clock is updated, it triggers several activation expressions in a specific order. The clock first triggers the audio input device, which reads a period of samples into a buffer and then activates any audio-input transforms in the program. All the time machines synchronized to this clock are then activated. Finally, the clock re-triggers the activation expression of the audio output device, thus completing the loop. By ordering the activations in this manner, we guarantee that in each period we first receive audio input, then perform audio processing that may depend on the input, and finally send the results to the audio output.

All the transforms that are connected to time-machines synchronized to same clock form a *synchronous dataflow graph*. Activation expressions in transforms of a synchronous graph are guaranteed to occur exactly once each period of the clock to which they are synchronized. Moreover, the order in which they are executed is fixed within a period. Because OSW allows multiple audio devices and clock sources as described above, several synchronous dataflow graphs that run at different sample rates and buffer sizes are supported. For example we may want to use an audio input running at a low sample rate (e.g., 11025Hz) for pitch detection, and send the resulting pitch values (i.e., as synchronous signals or asynchronous events) to a synthesizer that generates samples at a higher rate (e.g, 44.1Khz). Although the guarantees of synchronicity and fixed order of execution remain true within the synchronous graphs of individual channels, interaction between graphs requires explicit synchronization with one clock, usually the master clock. Otherwise, the interaction is asynchronous and non-deterministic.

4 Discussion

We have implemented and tested OSW on Intel-based PC's running the Windows NT and 98 operating systems. We are currently porting OSW to the Linux operating system, Macintosh computers running MacOS and SGI workstations. OSW has been used successfully in live musical performance situations [10].

5 Acknowledgements

We gratefully acknowledge the NSF Graduate Research Fellowship Program for their support of this research. We would also like to thank David Wessel and Lawrence A. Rowe, directors of CNMAT and the Berkeley Multimedia Research Center, respectively, for their support; and Roger Dannenberg from the School of Computer Science at Carnegie Mellon University for his constructive feedback.

References

1. J. Davis et al. "Heterogeneous Concurrent Modeling and Design in Java". Technical Report UCB/ERL M98/72, EECS, University of California, November 23, 1998 1998. `http://ptolemy.eecs.berkeley.edu`.
2. D. Zicarelli. "An Extensible Real-Time Signal Processing Environment for Max". *International Computer Music Conference*, pages 463–466, Ann Arbor, MI, 1998. ICMA.
3. T. Veldhuizen. "Scientific Computing: C++ Versus Fortran". *Dr. Dobb's Journal*, 22(11):34, 36–8, 91, 1997. Article Miller Freeman.
4. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 3rd edition, 1997.
5. E. A. Lee and T. M. Parks. "Dataflow Process Networks". *Proceedings of the IEEE*, 83(5):773–799, 1995.
6. E. Brandt and R. Dannenberg. "Low-Latency Music Software Using Off-the-Shelf Operating Systems". *International Computer Music Conference*, pages 137–140, Ann Arbor, MI, 1998. ICMA.
7. E. Clarke. "Rhythm and Timing in Music". Diana Deutsch, editor, *The Psychology of Music*, pages 473–500. Academic Press, San Diego, 1999.
8. M. Tsuzaki and R. D. Patterson. "Jitter Detection: A Brief Review and Some New Experiments". A. Palmer, R. Summerfield, R. Meddis, and A. Rees, editors, *Proceedings of the Symposium on Hearing*, Grantham, UK, 1997.
9. R. Dannenberg. "Real-Time Scheduling and Computer Accompaniment". Max Matthews and John Pierce, editors, *Current Research in Computer Music*. MIT Press, Cambridge, MA, 1989.
10. A. Chaudhary. "Spin Cycle / Control Freak", 1999. Live performance at CNMAT/CCRMA Spring 1999 Concert Exchange. Center for New Music and Audio Technologies, Berkeley, CA, May 15 1999.