Exploiting Parallelism in Real-Time Music and Audio Applications

Amar Chaudhary, Adrian Freed, and David Wessel

University of California, Berkeley, CA, USA

Abstract. We introduce a scalable, extensible object-oriented system developed primarily for signal processing and synthesis for musical and multimedia applications. The main performance issue with these applications concerns functions of discrete-time. Novel techniques exploit fine-grain parallelism in the calculation of these functions to allow users to express them at a high-level in C++. New scheduling strategies are used to exploit symmetric multiprocessors with emphasis on special hard real-time constraints.

1 Introduction

OSW, "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 [1]. OSW is a "dataflow programming language," similar to Ptolemy [2] and Max/MSP [3] in the signal-processing and computer-music communities, respectively. In OSW, components called *transforms* are connected to form dataflow networks called *patches*. OSW is also an "object-oriented" language in which transforms are instances of classes that specify their structure and behavior. OSW allows users to develop at multiple levels including visual patching (as seen in Fig. 1), high-level C++ and scripting. OSW includes a large set of standard transforms for basic event and signal processing. which 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 straightforward to add new data types as well.

2 Exploiting Fine-Grain Parallelism in OSW

The recently completed standardization effort for C++ introduced several new features, many of which directly address efficiency issues. 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.

S. Matsuoka et al. (Eds.): ISCOPE'99, LNCS 1732, pp. 49–54, 1999.

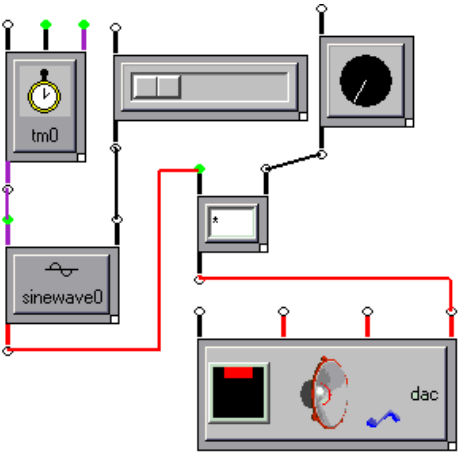

Fig. 1. A patch that plays a pure tone with varying frequency and amplitude

2.1 The Externalizer

OSW includes 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.

A transform is specified 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. An activation expression is a piece of C++ code that is executed when inlets or state variables are modified. It is specified by the variables that will trigger this activation, whether it should occur immediately or be delayed by a certain amount of time, and the code that should be executed. 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	
Inherited	SampleRate	float	44100.0
	NumberOfSamples	int	128

Activation Expression activation1, depends on timeIn, no delay

```
samplesOut = sin(TWOPI * frequency * timeIn);
```

The activation expression looks like a continuous function of time. However, as we will discuss in the next section, it is actually computing a sequence of samples from a discrete time variable, timeIn. The state variables NumberOfSamples and SampleRate are *inherited* from a more general class of *time-domain transforms* that manipulate time-domain samples.

An Externalizer transform specification is automatically converted to a C++ class which is then compiled into a dynamic library. The new library will by automatically loaded when the transform is first instantiated. Users can also specify new data types, which are converted to C++ `struct` definitions for use in transforms.

2.2 Optimizations on Functions of Time

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 in expressions that use `osw::vector<T>`[1] OSW's time data type and vector template class. Composition closure, functors and operator overloading are already well-known techniques in the numeric community, and are incorporated into several C++ numeric libraries [4]. 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. The overloaded `sin` function in the activation expression expands into a loop that fills a floating-point vector with values of the sine function over a regular sampling interval:

```
Samples temp; float t; int i;
for (t = timeIn->prev_time, i = 0;
     t < timeIn;
     t += timeIn->sampling_interval, ++i ) {
   temp[i] = sin(TWOPI * frequency * t);
}
samplesOut = temp;
```

In this example, `prev_time` is the previous value assigned to timeIn. The difference between the previous and current values of timeIn, divided by NumberOfSamples, is `sampling_interval`. This implementation has the virtue of simplicity, but is too slow for real-time work since the 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

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

will run on. The first step is to change the expansion of `sin` to more honestly reflect the fact that we are really computing discrete-time sequences:

```
for (i = 0; i < NumberOfSamples; ++i) {
   temp[i] = sin(TWOPI * frequency
      * (timeIn->prev_time + i * timeIn->sampling_interval));
}
```

Noting that the sequence $s_0 = 1, \ldots, s_n = s_{n-1}e^k$ computes e^{kn} for real and complex values of k, we can optimize computation of sinusoids using the imaginary result of Euler's identity $e^{i\theta} = \cos\theta + i\sin\theta$. Further noting that k, which is the product of 2π, the frequency and sampling interval, is a loop invariant, it can be calculated once prior to the loop.

```
float factor = exp(complex<float>(0.0,TWOPI * frequency
                  * timeIn->sampling_interval)));
for (i = 0; i < NumberOfSamples; ++i) {
     temp[i] = imag(timeIn->previous_sample * factor);
     timeIn->previous_sample = temp[i];
}
```

The property `previous_sample` has been added to the `Time` class to save the previous sample between activations. The inner loop of the expression has been essentially reduced to a single multiply operation via overloading, inlining and standard compiler optimizations. 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. Synchronization primitives (i.e., locks) are included to protect against nondeterministic 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 process 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. Examples of such chains include several channels of audio, or synthesizers with multiple voices (i.e., polyphony). If a transform has multiple outlets, the connections to each outlet will start new chains, so any activation expressions triggered by these connections will be added to a queue instead of being executed directly. Given N processors, we instantiate N threads for processing the queued expressions. Each thread executes the following loop *ad infinitum*:

loop
  pop an activation expression off the queue and execute it.
end loop

Low-priority tasks such as deallocation of dynamic memory are executed when there are no activations waiting to be scheduled. Higher-priority non-real-time tasks, such as handling user-interface events, are scheduled with a coarser granularity specified in a state variable. Additional threads are needed for handling some asynchronous input devices, such as MIDI input ports. 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 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:

```
while(SamplesInQueue()>TargetLatency-SampleBufferSize){ Wait();}
FlushSamples();
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.

Real-time audio contraints require that synchronicity guarantees be added to the process network scheduling described in the previous section. All the transforms that are connected to time-machines synchronized to the same clock must be executed exactly once each clock period. Because OSW allows multiple audio devices and clock sources with different sample rates and periods, it may be necessary to maintain several separate guarantees of synchronicity for each clock source.

4 Discussion

OSW runs on PC's running Windows NT/98 or Linux, and is being ported to SGI workstations. OSW has been used successfully in live musical performances [10]. See `http://www.cnmat.berkeley.edu/OSW` for more information.

Acknowledgements

We gratefully acknowledge the NSF Graduate Research Fellowship Program for their support of this research. We would also like to thank Lawrence A. Rowe, director the Berkeley Multimedia Research Center, for his support, and Matthew Wright for his contributions to the early design.

References

1. A. Chaudhary, A. Freed, and M. Wright. "An Open Architecture for Real-time Audio Processing Software". *107th AES Convention*, New York, 1999. 49
2. J. Davis et al. "Heterogeneous Concurrent Modeling and Design in Java". Technical Report UCB/ERL M98/72, EECS, University of California, November 23, 1998. `http://ptolemy.eecs.berkeley.edu`. 49
3. D. Zicarelli. "An Extensible Real-Time Signal Processing Environment for Max". *Int. Computer Music Conf.*, pp. 463–466, Ann Arbor, MI, 1998. 49
4. T. Veldhuizen. "Scientific Computing: C++ Versus Fortran". *Dr. Dobb's Journal*, 22(11):34, 36–8, 91, 1997. 51
5. E. A. Lee and T. M. Parks. "Dataflow Process Networks". *Proc. IEEE*, 83(5):773–799, 1995. 52
6. E. Brandt and R. Dannenberg. "Low-Latency Music Software Using Off-the-Shelf Operating Systems". *International Computer Music Conference*, pp. 137–140, Ann Arbor, MI, 1998. ICMA. 53
7. E. Clarke. "Rhythm and Timing in Music". Diana Deutsch, editor, *The Psychology of Music*, pp. 473–500. Academic Press, San Diego, 1999. 53
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, *Proc. Symp. on Hearing*, Grantham, UK, 1997. 53
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. 53
10. A. Chaudhary. "Two-Tone Bell Fish", April 25 1999. Live performance at CNMAT/CCRMA Spring 1999 Concert Exchange. Stanford University. 54