KIWI: A PARALLEL SYSTEM FOR SOFTWARE SOUND SYNTHESIS

William Franklin Walker
Department of Computer Science
University of Illinois at Urbana-Champaign

Kiwi is a software synthesis program which takes advantage of two innovations; object-oriented programming and parallel processing. The C++ programming language provides a clear framework for constructing Kiwi's components, while simultaneously providing an intuitive user interface. Multiprocessors allow near linear speedup for synthesizing most musical textures and linear speedup for reverberating any musical texture. Interesting issues raised while implementing Kiwi include optimizing scheduling algorithms, trading speed against storage, and parallelizing reverberation.

Early software synthesis programs, such as MUSIC 4BF and MUSIC 5 were written in FORTRAN and assembly language. More recent systems have been written in C. These systems are all firmly based on procedural, block-structured languages and single processor computers.

Two contemporary developments in computing seem particularly applicable to software synthesis. One is the emergence of commercially available parallel computers like the Alliant, Encore, and Sequent. Parallel architectures can greatly speed the generation of samples. The second is the growth of object-oriented programming languages like Smalltalk-80 and C++. These languages permit programmers to write and maintain large programs by encouraging modularity and good design practice.

Kiwi is a software synthesizer which parallelizes the task of sample computation from these three assumptions:

1. Music can be represented as a time-ordered list of events, or notes. Although there are other possible representations, the division of music into notes and timbres is a paradigm most composers accept.

2. The sound of a note is entirely independent of the sounds of other notes. This is true of monophonic instruments, and many polyphonic instruments can be modeled as collections of monophonic instruments.

3. The samples for a given note can only be computed in strict order. In practice, synthesis algorithms are given as recurrence relations in which the next sample depends on state variables. Thus, the value of the function at time t can only be computed by first computing the value of the function for all the times preceding t.

These assumptions shape Kiwi's structure. The first allows one to divide music synthesis into discrete tasks. From the second assumption it follows that no note interdependencies
need be considered when scheduling notes. Since the third assumption constrains us to compute each note's samples in strict order, no two processors should compute samples for the same note.

Kiwi was written in C++ for UNIX ¹ on the Encore Multimax. Kiwi also runs under CHOICES, an object-oriented operating system being developed at the University of Illinois.

```
  Score
    ↓
  ScoreList
    ↓
  Scheduler
    ↓
  Processor
    ↓
  Collector
    ↓
  Processor
    ↓
  Samples
```

This diagram shows the flow between Kiwi objects. The composer's score is converted into a linked list of Notes, which are stored in the ScoreList. Then several Processors are created. The number of Processors created is set by the user, and depends on the size and usage load of the user's computer. Tasks are assigned to these Processors by a Scheduler. The Scheduler constructs an instance tree of Sound objects for each note. The instance tree implements the timbre specified by the composer. When each Task terminates, the Processor passes it to the Collector, which sums samples from each Note and builds a sound file. A detailed description of these Kiwi objects follows:

**Sounds**: The Sound class is the root of a hierarchy of classes, each of which inherits from its parent. This hierarchy is a library of building blocks which can be combined to specify complex timbres. Each subclass of Sound defines the function next sample. This consistent interface allows objects to pass samples while hiding the algorithms used to generate them. Sounds are either dependent or independent. An independent sound can compute samples from its state variables alone. A lookup table oscillator uses only its waveform table, phase pointer, and phase increment to compute its next sample. Dependent Sounds use other Sounds to compute their samples. For instance, a band pass filter has a pointer called source which points to the Sound object being filtered. When Sounds are connected together, they an instance tree. Dependent Sounds are the interior of the tree while independent Sounds are the leaves of the tree.

**Task & Processor**: in many operating systems, a processor sits idle until the scheduler gives it a task. Kiwi reverses this approach; when a Processor is idle, it requests a Task from the Scheduler. A Kiwi Task has the form "compute samples for Note x starting at time y and ending at time z." The Processor computes these samples and stores them in the Task buffer, which is sent to the Collector.

**Collector**: Kiwi divides the piece into grains². These grains are of fixed duration, independent of the score. The Collector holds one grain of samples in main memory. Within that grain, samples can be accessed in any order. All the samples for one grain must be computed before samples for the next grain will be accepted. As soon as all the

¹UNIX is a trademark of AT&T Bell Laboratories
²Kiwi grains should not be confused with the technique known as granular synthesis.
samples for a given grain are computed, the Collector signals the Scheduler, and
computations for the next grain begin.

Scheduler: The Scheduler maintains an internal clock, myclock, and a subset of the
score list (the "current chord") containing all the Notes sounding at time myclock. When
a Processor requests a new Task from the Scheduler, the Scheduler does one of two things.
If there is a note in the current chord that has not been scheduled, the Scheduler
constructs a new Task representing the portion of that note which falls in the current
grain. Thus, a note spanning several grains will require several Tasks. If all of the Notes
in the current chord have been scheduled, then the Scheduler waits for the Collector to
signal and moves myclock ahead to the next grain. Then, the current chord's sublist is
updated to reflect the new clock. Finally, the Scheduler creates a Task for the first note in
the new chord.

Reverberation: The Kiwi reverberator parallelizes reverberation calculations by
exploiting the linearity of most audio signal processing algorithms. Reverberation can be
expressed entirely in terms of filters, which can be expressed linearly in the digital
domain. Since a reverberator is linear, it is commutative with addition: the reverberation
of a sum of notes is equal to the sum of the notes reverberated separately.

The Kiwi reverberator divides the entire sound file into equal segments such that the
size of a segment multiplied by the number of processors fits into memory. Each processor
takes a segment from the file, reverberates it, and stores it.

Reverberators continue to produce non-zero output for some time after their inputs
have dropped to zero. Filters that have this property are said to "ring." Ringing simulates
the sound in a reverberant room gradually dying away. If a processor computes
reverberation for a segment of the sound file, there will be a set of samples representing
the ringing which should be added to the beginning of the next segment. This overlap is
added to the next segment after that segment has been reverberated.

The Kiwi Scheduler performs well for a variety of musical textures, because the
parallelism is limited only by the number of different notes within a grain. Whether they
overlap or not is immaterial. Thus, even monophonic passages are parallelized whenever
multiple notes fall in the same grain.

The obvious way to measure Kiwi's performance is to measure the effect of adding
more processors on the total running time. Optimally, the running time for a given piece
using n processors should be the running time for using one processor divided by n. Kiwi
was run with three different score files using different numbers of processors. The total
running time was divided by the duration of the sound generated, yielding a performance
ratio:

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>20.1</td>
<td>10.9</td>
<td>7.9</td>
<td>6.5</td>
<td>5.0</td>
<td>5.2</td>
</tr>
<tr>
<td>Test 2</td>
<td>14.5</td>
<td>7.6</td>
<td>5.5</td>
<td>4.6</td>
<td>4.9</td>
<td>4.6</td>
</tr>
<tr>
<td>Test 3</td>
<td>18.0</td>
<td>10.3</td>
<td>10.3</td>
<td>10.1</td>
<td>10.1</td>
<td></td>
</tr>
</tbody>
</table>

Although most of the results are encouraging, Kiwi performed poorly on the third
test. The minimal running time was achieved with only four processors; even quadrupling
that number to sixteen processors made no significant difference. The third test score was
generated by a simple fractal algorithm. Durations in the resulting score range from
Several hundredths of a second to several dozen seconds. The plateau in Kiwi's performance reflects the fact that not all Tasks represent equal work. Each grain in the third test score has at least one Task which represents a note lasting the entire grain, requiring much computation, and many other Tasks for short notes which require little computation. The overall speed is governed by the speed of these long, computationally intensive Tasks.

The reverberator's performance does not depend on the musical texture, since it operates on the completed sample file. The only factors governing its performance are the complexity of the reverberation algorithm and the number of processors used. Thus, its performance is linear in the number of processors:

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test 1</td>
<td>42.1</td>
<td>22.8</td>
<td>14.8</td>
<td>11.2</td>
<td>9.4</td>
<td>7.5</td>
</tr>
</tbody>
</table>

Kiwi has changed dramatically since work began in the summer of 1988, and will continue to serve as a platform for testing ideas about scheduling and sound synthesis. Several improvements are planned for the Scheduler. The Scheduler should process requests for new Tasks in parallel. It should ensure that large Tasks are scheduled before small Tasks, since the large Tasks can dominate the speed of computation. In addition, the Sound hierarchy should be expanded, and tools should be provided to design Kiwi orchestras automatically, freeing the composer from the need to write C++ code in most cases.

The design objective of this project was to build a software sound synthesizer that was both elegant and efficient. Kiwi takes advantage of two innovations: object-oriented programming and parallel processing. The C++ programming language provides a clear framework for constructing Kiwi's components, while simultaneously providing an intuitive user interface. Multiprocessors allow near linear speedup for synthesizing most musical textures and linear speedup for reverberating any musical texture. These two new technologies allowed Kiwi to achieve its design goal, which in turn gives the composer a better tool for making music.

References

