LAZY EVALUATION IN MICROSOUND SYNTHESIS

Hiroki Nishino
Imagineering Institute, Malaysia &
Chang Gung University, Taiwan
hiroki.nishino@acm.org

Adrian David Cheok
Imagineering Institute, Malaysia &
City University London, United Kingdom
adrian@imagineeringinstitute.org

ABSTRACT

The microsound synthesis framework in the LC computer music programing language integrates objects and library functions that can directly represent microsounds and related manipulations for microsound synthesis. Together with the mechanism that enables seamless collaboration with the unit-generator-based sound synthesis framework, such abstraction can help provide a simpler and terser programing model for various microsound synthesis techniques.

However, while the microsound synthesis framework can achieve practical real-time sound synthesis performance in general, it was observed that temporal suspension in sound synthesis can occur, when a very large microsound object beyond microsound time-scale is manipulated, missing the deadline for real-time sound synthesis.

In this paper, we describe our solution to this problem. By lazily evaluating microsound objects, computation is delayed until when the samples are actually needed (e.g., for the DAC output), and, when performing the computation, only the amount of samples required at the point is computed; thus, temporal suspension in real-time sound synthesis can be avoided by distributing the computational cost among the DSP cycles. Such a solution is beneficial to extend the application domains of the sound synthesis framework design beyond microsound synthesis towards more general sound synthesis techniques.

1. INTRODUCTION

Today, microsound synthesis techniques [1] already constitute an important part of digital sound synthesis techniques for musical creation, being used for both non-real-time and real-time sound synthesis. Unlike many other sound synthesis techniques that conceptualize sounds as functions of time, microsound synthesis conceptualizes the sound as a composition of many short sound particles that overlap-add onto each other. Such a significant conceptual difference led to the question if the traditional unit-generator concept [2, p.89], which describes a sound synthesis algorithm by software modules that stream sample data to each other, is still appropriate also for microsound synthesis. While there have not been many examples, some previous works investigate more suitable software abstractions for microsound synthesis for this reason.

The LC computer music programming language [3] that we developed is one of the most recent examples of this kind, which takes microsound synthesis techniques into account in the software abstraction. The microsound synthesis framework in the LC language [4] significantly differs from existing unit-generator-based synthesis frameworks and integrates objects and library functions that can directly represent microsounds and related manipulations in microsound synthesis. Such abstraction contributes to describing various microsound synthesis techniques much simpler and terser in comparison with many existing unit-generator languages and can realize microsound synthesis in real-time.

Yet, as we previously described in [4], temporal suspension of real-time sound synthesis can be observed in certain situations; as LC's microsound objects are arrays of sample values (with useful methods) in nature, when manipulating a microsound object of a very large size, far beyond the microsound time-scale, the deadline for the real-time sound synthesis can be missed, causing temporal suspension of the audio output audible to human ears. We also described that this issue has normally not been observed in most practical situations when the sizes of microsound objects stay reasonably within the microsound time-scale and microsounds are manipulated sporadically as assumed by the sound synthesis framework. Yet, it is still desirable to avoid such temporal suspensions by large microsound objects to extend the application domain to more general applications beyond microsound synthesis.

In this paper, we propose a solution to this problem by adopting lazy evaluation to the microsound synthesis framework. Instead of eagerly evaluating the results of manipulations right when they are performed, the evaluation is delayed until when the result is actually required. Also, the evaluation takes place only for the number of samples required at that point, rather than for the entire microsound object; thus, by the adoption of lazy evaluation to the manipulation of microsound objects, the pause time can be significantly reduced and temporal suspension can be avoided in most practical situations, even when manipulating large microsound objects.

To assess the actual performance efficiency without being influenced by other factors in the language implementation (e.g., memory allocation, garbage-collection, or task-switching), we implemented a testing software framework and measured the performance efficiency in C++. The results showed a significant reduction of pause time, just as expected.

Such adoption of lazy evaluation to microsound objects, not only solves the issue of the temporal suspensions in most practical situations, but can also contribute to making the microsound synthesis framework more stable at runtime, by distributing the computational cost beyond one DSP cycle; this is quite favorable towards more general applications of the sound synthesis framework design, certainly beyond microsound synthesis.

2. RELATED WORK

2.1 Microsound Synthesis

Microsound synthesis was first brought to the practice of computer music around the early 1970s[1]. Since then, various sound synthesis techniques of the kind have been developed. This includes granular synthesis [5], formant wave-function (FOF) (from *function d'onde formantique)* [6], FOF synthesis [7], and waveset synthesis [8], all these belonging the family of microsound synthesis.

While many other synthesis techniques that conceptualize a sound as a function of time (such as additive synthesis, subtractive synthesis and FM synthesis), microsound synthesis conceptualizes sound quite differently. Generally speaking, in microsound synthesis, the entire sound output is composed of many short sound particles (i.e., microsounds) that overlap-add onto each other. The duration of such short sound particles extends from "the thread of timbre perception (several hundred micro seconds) up to the duration of a short sound object (~100 msec)" to "the boundary between the audio frequency range (approximately 20 Hz to 20 kHz) and the infrasonic frequency range (below 20 Hz)" [1, p.21].

In fact, Gabor, whose theory had a significant influence on the origins of microsound synthesis, already contrasted his theory to "the orthodox method of analysis … [which] starts with the assumption that the signal is a function (t) of time t" [9].

2.2 Software Frameworks for Microsound Synthesis

Such a theoretical difference as described earlier led to the question of whether the traditional unit-generator concept is appropriate for microsound synthesis[2].

Typically, in unit-generator languages, microsound synthesis techniques are normally realized by unit-generators, which encapsulate microsound synthesis techniques within, or by implementing microsounds as note-level sound objects and scheduling them with the user program.

Yet, some computer music researchers discuss that such implementations may not be appropriate for microsound synthesis. In [10], Brandt argues that "Music-N languages like Csound [11]," which are typical unit-generator languages, "are too limited" and not appropriate for FOF synthesis, since "the stream [of grains in FOF synthesis] is irregularly timed, and a grain is a sequence of samples". Brandt also puts forth that the unit-generator concept is a "black-box primitive" in computer music language in nature and is problematic as "if a desired operation is not present, and cannot be represented as a composition of primitives, it cannot be realized within the language" [12, pp.4-5].

In [4], we too maintained that the unit-generator concept may not be truly beneficial for end-users even when note-level objects and scheduling algorithms are utilized for the implementation of a certain microsound synthesis technique and provided as a library function, especially when considering creative explorations by users. Even though the complicated implementation is hidden inside the library function, if the user needs to alter the sound synthesis algorithm beyond what is provided by the function, the user has to modify the hidden implementation within the library function; such a situation is clearly not desirable to support the activities of exploratory understanding and exploratory design[3].

Based on such considerations, researchers and developers have been investigating alternative software abstractions that are suitable for microsound synthesis. For example, Bencina's object-oriented software framework design for a granular synthesizer includes the objects that directly represent grains (microsounds). Brandt's Chronic language is another example [12]. Chronic is a computer music language built upon the OCaml language [13]. Brandt proposed the 'temporary type constructor' concept, which introduces "a relation to a one-dimension axis, which we call time" to type constructors[4]. In doing so, for example, the audio stream of granular synthesis can be defined by the type of "*Sample vec event ivec.*"; each grain is represented by *Sample vec* (a vector of samples with finite size) and it is scheduled with a timestamp (*event*) in the infinite length stream of such events (*ivec*). With this sort of abstraction, Chronic can provide the direct access to low-level sample data, which does not normally exist in unit-

[1] For instance, one of the earliest examples of microsound synthesis was the implementation of ascynronous granular synthesis in MUSIC-V by Curtis Roads in 1974. Roads [1, p.110].

[2] If interested, see [2] for the detailed discussion by Nishino et al.

[3] Blackwell and Green list such activities as sketching; design of typography, software, etc.; other cases where the final product cannot be envisaged and has to be 'discovered' as the examples of exploratory design and discovering structure of algorithm, or discovering the basis of classification as the examples of exploratory understanding [13]

[4] "A type constructor builds complex types from simpler ones. For example, C has the "pointer to…" type constructor and we can write this as "α *pointer,"* where α is a free type variable which might be. for example, int" [12, p.7]

generator languages; this feature is beneficial for peforming various microsound synthesis techniques just within the language, without the help of 'native' modules written in C++.

However, Bencina's software framework is mainly designed for stand-alone synthesizer software and Brandt's Chronic language is a non-real-time computer music language and significant reconsideration is required to be adopted for real-time sound synthesis, because of its acausal behavior [12, p.77].

2.3 The Microsound Synthesis Framework in LC

In contrast, our LC language is designed with the sound synthesis framework deemed appropriate for real-time sound synthesis. Sharing the same interest with Bencina's granular synthesis framework and Brandt's Chronic language for investigating in alternative software design besides the traditional unit-generator concept, the microsound synthesis framework in the LC computer music language was designed with objects and library functions that can directly represent microsounds and related manipulations for microsound synthesis.

```
//create a new Sample object from the buf no. 0.
LoadSndFile(0 "/sound1.aif");
var snd = ReadBuf(0, 256::samp);

//create another by generating a window.
var win = GenWindow(512::samp, \hanning);

//create Samples objects by the method calls.
var grain    = snd->applyEnv(win);
var halfAmp  = snd->amplify (0.5);
var octup    = snd->resample(snd.size / 2);
var reversed = snd->reverse();

//convert a Samples obj to a SampleBuffer obj.
var sbuf = snd->toSampleBuffer();
//convert it back to a Samples obj.
var snd2 = sbuf->toSamples();

//create a new SampleBuffer by the 'new' operator.
var sbuf2 = new SampleBuffer(128);
```

Figure 1. *Samples* and *SampleBuffer* objects in LC.

```
//indexed-access to a SampleBuffer object.
var sb = new SampleBuffer(256);
for (var i = 0; i < sb.size; i+=1){
  sb[i] = i * 2
}

//indexed-access to a Samples object.
//Samples is read-only (immutable)
var snd = sb->toSamples();
for (var i = 0; i < snd.size; i +=1){
  println("snd[" .. i .. "]=" .. snd[i]);
}
```

Figure 2. Example of indexed access in LC.

In the microsound synthesis framework, two objects, *Samples* and *SampleBuffer* directly represent microsounds. While the former is immutable and the latter is mutable, the methods to convert between these two objects are provided. The *Samples* object is mainly used to manipulate and schedule microsounds. Figure 1 shows various methods to create *Samples* objects and Figure 2 displays the examples of index access to the *Samples* and *SampleBuffer* objects.

In LC's microsound synthesis framework, microsound synthesis is performed with these microsound objects together with various methods and library functions. Figure 3 and Figure 4 depict examples of microsound synthesis in LC, just as described in [3] and [4].

```
01 //create a SampleBuffer and fill it with 256 samp
02 //sine wave * 4 cycles.
03 var PI = 3.14159265359;
03 var sbuf = new SampleBuffer(1024);
04 for (var i = 0; i < sbus.size; i+=1){
05   sbuf[i] = Sin(PI * 2 * (i * 4.0 / sbuf.size);
06 }
07
08 //create a grain, apply an envelope, resample it.
09 var tmp = subf->toSamples();
10 var win = GetWindow(1024:samp, \hanning);
11
12 var grain = tmp->applyEnv(win)->resample(440);
13 grain = grain->amplify(0.25);
14
15 //perform granular synthesis for 5 sec.
16 within(5::second) {
17   while(true){
18     WriteDAC(grain);
19     now += grain.dur / 4;
20   }
21 }
```

Figure 3. Example of synchronous granular synthesis in LC [3, p.132].

```
//create an array to store pregenerated grains
var grains = new Array(100);

//generate grains with 400-500 Hz sine waves
var win = GetWindow(512::samp, \hanning);
for (var i = 0; i < grains.size; i += 1){
  //use a unit-gen object to create a Samples obj.
  var src = new Sin~(i + 400);
  var tmp = src->pread(win.dur);
  var grn = tmp->applyEnv(win);
  grains[i] = grn;
}

//perform granular synthesis for 5 sec.
within(5::second) {
  while(true){
    var idx = Rand(0, grains.size – 1);
    PanOut(grains[idx]);
    now += grains[idx].dur / Rand(0.5, 2);
  }
}
```

Figure 4. Example of granular synthesis with pregenerated grains [4].

Figure 3 is an example of simple synchronous granular synthesis [5] in LC. As shown, the *Samples* and *SampleBuffer* objects are used to represent microsounds, and manipulations can be directly applied to these objects (lines 12-13) by method calls (see [3, p.132] for more details of the code). Note that as the *Samples* object is immutable, it can be reused and rescheduled even when the same object may overlap at a certain point in time (line 17-20), without any extra care. Generally speaking, the sound object in unit-generator languages (such as *instrument* in

[5] Synchronous granular synthesis is a kind of granular synthesis, in which "sounds results from one or more streams of grains" (i.e. stream(s) of microsounds). "Within each stream, one grain follows another, with a delay period between the grains. Synchronous means that the grains follow each other at regular intervals" [1, p.93].

Csound and *synth* in SuperCollider [14]) cannot realize such self-overlapping, because each unit-generator must maintain its own internal current status that changes as sound synthesis is performed.

In contrast to synchronous granular synthesis, asynchronous granular synthesis "scatters the grains over a specified duration within regions inscribed on the time-frequency plane" [1, p.96]. Figure 4 is an example of asynchronous granular synthesis with the grains made from the sine wave of frequency between 400-500 Hz and irregular intervals. In this example, since the grains are pre-generated beforehand and only scheduling is performed in actual sound synthesis, significantly enhanced performance efficiency can be achieved. Such abstraction of the microsound synthesis framework in LC permits describing various microsound synthesis techniques more tersely and simpler versus existing unit-generator languages (see [3] and [4] for more details).

2.4 Lazy Evaluation and Digital Sound Synthesis

2.4.1 Lazy Evaluation

Lazy evaluation is a method to evaluate programs, often seen in functional programing languages. The list of the languages that have lazy evaluation in the language specification includes Haskell [15], OCaml [16], Scala [17], and others. In a *lazy language*, a program "will not evaluate any expression unless its value is demanded by some other part of computation", whereas, in a *strict language*, a program "evaluate[s] each expression as the control flow of the program reaches it[6]" [18, p.322].

While there are not many examples, certain computer music languages adopt lazy evaluation for sound synthesis. The following sections describe such languages.

2.4.2 The Fugue Computer Music Language

Fugue [19], a computer music language developed by Dannenberg et al. as an internal domain-specific language built on XLISP [20], is an early example of computer music language applying lazy evaluation in sound synthesis.

```
01 (setf Mysound (sfload "mysound")
02 (setf Demo (scale 2.0 (seq (cue Mysound)
03                            (cue Mysound))))
04 (play Demo)
```

Figure 5. A simple example of sound synthesis in the Fugue language [19].

Figure 5 portrays a simple example code in the Fugue language, as described by Dannenberg et al. in [19]. First, the variable *Mysound* is bound to a sound stored in the file "mysound" (line 01). Line 02 creates a score consisting two copies of *Mysound* scaled by 2.0 and sets it to the variable *Demo*. However, Fugue does not evaluate this score at this point and no computation is performed. In line 03, the *play* function call forces the evaluation to produce the sample output. When this computation is performed, the computed samples are memorized in *Demo*.

Nevertheless, unlike many other lazy languages, *Fugue* does not memoize the result of intermediate computations by default [21]; lazy evaluation is employed rather as a technique to eliminate unnecessary memory allocation and signal copying to form intermediate results for the improvement of the performance efficiency [21]. Fugue was significantly extended later and renamed *Nyquist*. Nyquist also performs lazy evaluation, yet approaches sound synthesis incrementally by block processing so that it can reduce the required memory space [22], while Fugue allocates the enough memory space for the entire result and computes one-at-a-time. The intermediate computed results are not memoized by default also in Nyquist [21].

2.4.3 The Chronic Computer Music Language

Chronic developed by Brandt [12], which is an internal domain-specific language (DSL) built on OCaml for non-real-time sound processing, is another noteworthy example of a computer music language that adopted lazy evaluation for sound synthesis. As already described in Section *2.2 Software Framework for Microsound Synthesis*, Chronic has the type *'ivec'*, which is a vector of infinite size. Unlike strict languages, lazy languages can handle vectors of infinite size without any difficulty, as any value in the vector is not evaluated until it is actually required.

As seen in the example of granular sound synthesis ("*Sample vec event ivec*"), Chronic utilizes this feature to express an audio stream of infinite length, without modeling it as a data-streaming object (like unit-generators). Such a framework design fosters removal of the abstraction barrier[7] to low-level sample data, by avoiding the encapsulation of it and allowing direct access, even when handling the streaming of audio and event data.

3. DESCRIPTION OF OUR WORK

3.1 Temporal Suspension of Sound Synthesis

As described in Section *2.3: The Microsound Synthesis in LC*, the LC language provides microsound objects and related manipulations for microsound synthesis. This software design assumes that microsound synthesis techniques deal with fairly short sound particles and scheduling algorithms are performed sporadically.

Yet, while this assumption is practically justifiable in performing microsound synthesis techniques, when manipulating a very large *Samples* object beyond microsound time-scale, real-time sound synthesis can be temporarily suspended, because the computation is performed eagerly

[6] This evaluation strategy of strict languages is also often referred to as *eager evaluation.*

[7] Abstraction barriers "isolate different 'levels' of the system." "At each level, the barrier separates the programs … that use the data abstraction from the programs … that implement the data abstraction" [24, p.88]

in the current version. It can consume too much time to manipulate very large *Samples* objects, and may fail to meet the real-time deadline for sound synthesis.

For example, if each DSP cycle requires 256 frames of samples for the audio output under the 44.1 kHz sample rate, this samples corresponds to about 5.8 msec. Yet, if a large *Samples* object, like one consisting of 4,410,000 (= 100 sec/44.1kHz) samples, is manipulated during one DSP cycle, it easily consumes more than 5.8 msec. This can suspend the real-time sound synthesis for a while; while this type of situation is beyond what the microsound synthesis framework in LC assumes, it is still desirable to avoid such temporal suspension. Figure 6 shows a simple example that would bring about such temporal suspension in LC.

```
//read three second from the buffer No.0.
var snd = ReadBuf(0, 1::second);
//play it.
WriteDAC(snd);

now += 0.5::second; //0.5 sec wait.

//resample it to 600 * 44100 samples.
//this temporarily suspend real-time synthesis,
//as it consumes too much CPU time.
var tmp = snd->resample(600 * 44100);
```

Figure 6. Example to temporarily suspend the DSP.

3.2 Lazily Manipulating Microsound Objects

We adopted lazy evaluation to solve this problem of temporal suspension. As the evaluation of a microsound object is delayed until it is actually required (e.g., for the audio output or for the access to the samples within the microsound object) and only the required part of the microsound object is computed, leaving the rest of samples uncomputed, the pause time imposed during evaluation can be significantly reduced.

We implemented a simple software framework in C++ from scratch to evaluate how effective this technique can be in practice; as it is clearly more desirable to avoid the influence from any other factors (such as memory allocation and garbage collection) when measuring the performance efficiency, we opt to not directly integrate this technique into an existing language.

Figure 7 shows the excerpt of the definition of the abstract class for microsound objects. As is present in line 06, each samples within the *Microsound* instance can be accessed by the overloaded operator '[]'. To exclude the influence of memory allocation, which can consume a significant amount of the CPU time from the measurement of the performance efficiency, in this testing framework, all the memory was allocated with the constructor method[8], and the *init* method performs any other initialization required for sound synthesis. The *init* method also calls the *_init* method of the subclass.

```
01 class Microsound {
02 public:
03   Microsound(void);
04   virtual ~Microsound(void);
05   virtual int64_t   size(void) = 0;
06   virtual lc_sample operator[](int64_t index) = 0;
07   virtual void      init(void) final;
08 };
```

Figure 7. The base class for all the microsound classes.

```
//the sinewave class (The eager-evaluation)
//called when the instance is initialized.
void MSEagerSineWave::_init(void)
{
  double phase = 0.0;
  double phaseInc = 2.0 * PI * freq / gSampleRate;

  int64_t size = this->size();
  for (int64_t i = 0; i < size; i++){
    this->buf[i] = sin(phase) * amp;
    phase += phaseInc;
   if (phase > 2.0 * PI || phase < -2.0 * PI){
      phase = fmod(phase, 2.0 * PI);
    }
  }
  return;
}
```

Figure 8. Excerpt of the sinewave microsound class implementation (the eager-evaluation).

```
MSLazySineWave::_init(void)
{
  this->pahseInc = 2.0 * PI * freq / gSampleRate;
}
lc_sample MSLazySineWave::operator[](int64_t idx)
{
  int64_t bitmapIndex = index / MSBLOCK_SIZE;
  if (this->computedBlocks[bitmapIndex]){
    return this->samples[idx];
  }

  int64_t start = bitmapIndex * MS_BLOCK_SIZE;
  int64_t end   = start + MS_BLOCK_SIZE;
  if (end >= this>samples.size()){
    end = this->samples.zie();
  }

  double phz = fmod(phaseInc * start, 2.0 * PI);
  for (int64_t i = start; i < end; i++){
    this->samples[i] = (lc_sample)(sin(phz)* amp);
    phz += phaseInc;
  }
  this->computedBlocks[bitmapindex] = true;
  return this->samples[idx];
}
```

Figure 9. Excerpt of the sinewave miscrosound class implementation (the lazy-evaluation).

Figure 8 and Figure 9 present the excerpts from the implementation of the sinewave microsound class. Each figure shows the versions that perform eager evaluation and lazy evaluation, respectively. In Figure 8, before the instance is created[9], the *_init* method is called to fill the buffer to with the waveform of the sine wave at once. The computed samples can be accessed by the '[]' operator.

On the contrary, in Figure 9, the *_init* method of the lazy evaluation version only sets up the phase increment parameter, and does not compute any samples. Yet, when the indexed access is performed, it verifies whether the sample

[8] Based on the fact that the CPU time used for memory allocation is often less predictable and can consume significant amounts of time, real-time computer music software needs to take extra care, as discussed by Dannenberg and Bencina [24].

[9] To exclude the CPU time for the memory allocation from the performance evaluation, the required memory space was allocated before real-time sound synthesis began. The other components performing signal processing were executed during real-time synthesis.

value at the index is previously evaluated. If already evaluated, the method simply returned the memoized value. If not, it computes the sample value and return.

Note that one block of samples is computed altogether by block processing to improve performance efficiency. The internal buffers within the microsound objects are divided into a number of the blocks with fixed size, and each block is computed at once, when any access to a sample value within the block is made. The flag if the block is already computed or not is maintained in the separate bitmap (as observed in lines 08-11 of Figure 9). Since the std::vector<bool> uses only one byte for eight elements of the boolean type, each byte can manage the statuses of eight blocks.

While lazy evaluation can distribute the cost of the computation among the DSP cycles by computing on demand, such block processing also contributes to increasing computational efficiency when the computation is demanded; thus, the pause time in the creation of microsound objects can be significantly reduced.

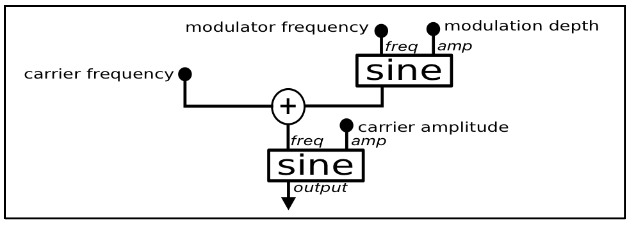

Figure 10. A simple FM synthesis instrument

```
//building a FM synth sound (carrier freq = 1000,
//modulator freq = 6, modulation depth = 15)
int64_t size = GetSampleRate() * 5;
mod   = new MSLazySineWave  (size, 6, 15);
cfreq = new MSLazySig       (size, 1000);

//add the carrier freq and the modulater output.
freq  = new MSLazyArithmetic('+', mod, cfreq);

//generating the final output samples.
amp   = new MSLazySig       (size, 1.0);
out   = new MSLazySineWave2 (freq, amp);
```

Figure 11. Performing FM synthesis with microsound objects.

MSEagerSig and MSLazySig:	is a microsound object with the constant values.
MSEagerWhiteNoise and MSLazyWhiteNoise:	is a microsound object filled with white noise.
MSEagerSineWave and MSLazySineWave:	is a microsound object filled with a sine wave. The frequency and amplitude parameters are specified by floating point values.
MSEagerSineWave2 and MSLazySineWave2:	is a microsound object filled with a sine wave. The frequency and amplitude parameters are given by other microsound objects.
MSEagerArithmetic and MSLazyAritmetic:	creates a new microsound objects. By performing an arithmetic operation (+,-,*,/) to two input microsound objects and
SUEnvelope:	creates a new microsound with the given envelope parameters.

Table 1. The list of the available unit-generators

Table 1 lists the microsound objects prepared for the assessment of the performance efficiency. While there are few objects, these are enough to perform basic tasks, such as additive synthesis and FM synthesis. For instance, a simple FM synthesis (as portrayed in Figure 10) can be carried out by combining these objects as seen in Figure 11.

4. PERFORMANCE MEASUREMENT

4.1 The Test Environment

As described in the previous section, the testing software framework was written exclusively in C++ from scratch, independently of any existing computer music software, so that we could exclude other factors regarding the language implementation as possible. We used the *clock* library function so that the exact CPU time spent for the sound synthesis would be measured without the influence of task-switching.

All the tests were performed on a Mac Book Air 2015 (11-inch, Intel Core i5 1.6GHz, 4GB Memory, OS X El Capitan). The code was compiled with the '-Ofast' option (the fastest aggressive optimization) with the Apple LLVM7.1 compiler. The I/O block size for the sound output was set to 256 samples, and the sample rate was set to 44.1kHz. The block size for the lazy evaluation version of microsound objects was set to 256 samples.

4.2 The Test Tasks

Table 2 outlines the test tasks for the evaluation. Each task was performed five times for both eager evaluation and lazy evaluation to measure the worst-case CPU time, the best-case CPU time, and the average CPU time. Each task generated the sounds of 10 sec, 30 sec, and 120 sec.

Task #1: White Noise with Ring Modulation	A white noise sound is scaled by a sine wave sound.
Task #2: Additive Synthesis	Additive synthesis consisting of four sine wave sounds and one envelope applied to the entire sound.
Task #3: FM Synthesis	A simple FM synthesis sound as described in Figure 10.

Table 2. The test tasks for the performance measurement.

4.3 The Test Results

Table 3 details the results of the test tasks. All the results in the table are in milliseconds. The numbers in the '*at the initialization*' section refer to the worst-case CPU time (max), the best-case CPU time (min), and the average CPU time in the DSP cycle when the microsound objects are initialized.

The numbers in the 'rest' section are the CPU times spent in the rest of the DSP cycles until the end of the sound. The total CPU time is the entirety of the CPU time spent finishing the evaluation of the microsound objects (the average of the five trials for each task).

	At the initialization			*The rest*			*Total CPU time (ms)*
	avg (ms)	*max (ms)*	*min (ms)*	*avg (ms)*	*max (ms)*	*min (ms)*	
The Fastest-Aggressive Optimization (-Ofast)							
Task 1: White Noise With Ring Modulation							
10 sec							
Eager	21.334	25.090	17.494	0.008	0.029	0.001	34.762
Lazy	0.088	0.092	0.081	0.029	0.105	0.012	50.518
30 sec							
Eager	63.685	72.217	55.735	0.009	0.115	0.002	112.537
Lazy	0.053	0.062	0.046	0.032	0.158	0.011	164.912
120 sec							
Eager	208.183	224.036	196.261	0.007	0.053	0.001	350.470
Lazy	0.077	0.092	0.055	0.038	0.192	0.010	794.457
Task 2: Additive Synthesis							
10 sec							
Eager	61.828	80.981	53.277	0.008	0.030	0.002	75.798
Lazy	0.089	0.107	0.052	0.083	0.489	0.033	142.842
30 sec							
Eager	201.302	220.587	179.687	0.008	0.053	0.001	241.545
Lazy	0.095	0.137	0.058	0.083	0.502	0.032	427.433
120 sec							
Eager	803.788	881.327	679.922	0.014	0.164	0.002	1092.172
Lazy	0.122	0.154	0.089	0.095	0.543	0.032	1954.035
Task 3: FM Synthesis							
10 sec							
Eager	27.320	33.822	24.116	0.008	0.143	0.002	41.423
Lazy	0.070	0.094	0.048	0.033	0.116	0.014	57.505
30 sec							
Eager	84.239	106.615	67.021	0.008	0.050	0.001	124.650
Lazy	0.070	0.095	0.050	0.034	0.161	0.014	173.607
120 sec							
Eager	352.669	406.257	288.634	0.013	0.284	0.002	621.765
Lazy	0.106	0.124	0.076	0.040	0.377	0.012	817.449

Table 3. The Test Results

5. DISCUSSION

5.1 The Evaluation of the Test Results

Overall, the test results indicated what was theoretically expected. In the eager evaluation version, as the sizes of the microsound objects became larger, the time costs associated with the evaluation at the initialization increased almost proportionally. For example, the average CPU time per DSP cycle (avg) observed for the FM synthesis task by eager evaluation was 27.320 msec for the 10 sec sound, 84.239 msec for the 30 sec sound, and 352.69 sec for the 120 second. The time cost after the initialization stayed constant regardless of the duration of the sound, because it only retrieved the sample data already computed at the timing of the initialization. As shown in Table 3, it ranged from 0.08 msec to 0.014 msec.

In contrast, lazy evaluation significantly diminished the pause time imposed by the manipulation of microsound objects, even when the sizes of the microsound objects were far beyond the microsound time-scale as in these test tasks. The CPU cost in each DSP cycle remained mostly constant regardless of the size of the microsound object. For example, the CPU time spent at the initialization ranged between 0.053 msec and 0.122 msec, and the average CPU time for the rest stays almost constant for each task (0.029 – 0.33 msec for task 1, 0.083-0.095 msec for task 2, and 0.033 – 0.040 msec for task 3). While it was observed that the total CPU time costs were equal to roughly 1.5-2 times as much as the eager evaluation version, the significant reduction of the pause time achieved by lazy evaluation is quite favorable for computer music applications. To meet the real-time deadline is the most important criterion for real-time sound synthesis.

Thus, the adoption of lazy evaluation in our technique significantly reduced the pause time and therefore the temporal suspension, as described in [3] and [4], can be avoided. While our original microsound synthesis framework in the LC language assumed only the use for microsound synthesis techniques, our lazy evaluation technique can aid in enlarging the potential application domain of his microsound synthesis framework, towards more general sound synthesis techniques.

5.2 The Difference from Existing Works

As discussed in the Related Work section, there are not many examples of the previous literature utilizing lazy evaluation for digital sound synthesis, and these existing works significantly differ from that which we presented here. Both Fugue and Chronic perform non-real-time sound synthesis. Fugue (and Nyquist) adopts lazy evaluation to reduce memory allocation to improve the overall performance efficiency and Chronic employs lazy evaluation to express a data stream as an infinite-length vector; in contrast, our work adopts lazy evaluation to reduce the pause time in real-time sound synthesis, by distributing the computational cost among the DSP cycles.

5.3 The Memory Usages

Memory usage is one of the issues needing to be discussed regarding our technique. Indeed, this is one of the reasons Nyquist utilizes block processing and does not memoize the intermediate results, as the available physical memory space was not as large as is common place today and the allocation of large memory can lead to frequent paging to/from the external storage, significantly damaging the performance efficiency. However, computer systems have much larger physical memory space nowadays, and the audio data is not overly large for fitting in the physical memory space in most situations. Moreover, while our current implementation holds intermediate results, if the computer music language has the garbage collection feature, the intermediate results unreachable from the program can be automatically released[10].

5.4 The Extended Discussion

One of the possible extension for this technique is to greedily evaluate the samples before they are actually needed. The samples can be computed in other threads in parallel with the audio thread. Generally speaking, when performing real-time sound synthesis, the audio thread periodically computes the audio output with a certain interval so that it can coordinate its computation with the progress of real

[10] If garbage collection utilizes reference counting [25] (or combine it with another garbage collection mechanism), such memory space can be released immediately when it becomes unreachable. This would make the reuse of the memory space allocated for microsound objects faster.

time. Such temporal behavior is important to realize interactive control in a computer music system.

However, while the sound synthesis is performed during this periodic computation in the audio thread in the current test environment, it is also possible to evaluate the sample values left uncomputed within microsound objects in background threads in parallel. This would not cause much damage to the temporal behavior of the audio thread, since it is not necessary to synchronize these threads for the audio computation, if microsound objects are immutable (as in LC's *Samples* object). Even when the audio thread and background threads compute the same block of samples and evaluate the samples data simultaneously, the computed results are identical because of immutability; hence, the evaluation can be performed in parallel without any problem and significant improvement of the computational efficiency can be expected with multithreading.

6. CONCLUSIONS AND FUTURE WORK

In the present work, we proposed a solution to the problem of temporary suspension of real-time sound synthesis as seen in the microsound synthesis framework in the LC language. By adopting lazy evaluation to manipulations of microsound objects, the pause time imposed by the manipulations of microsounds can be significantly diminished and, thus, temporary suspension can be avoided. Such a feature is quite favorable for utilizing the software abstraction of the microsound synthesis framework in more general applications beyond microsound synthesis. When combined with the reusability of previously computed samples (e.g., reusing the pre-generated microsounds), a significant improvement can be expected in the performance efficiency in real-time sound synthesis, which existing unit-generator languages can hardly emulate.

For the future research, we are planning to extend the technique, as alluded to earlier, to evaluate samples that are still not yet computed, greedily in multithreads, for further improvement in the performance efficiency.

Acknowledgments

We gratefully thank Prof. Roger Dannenberg for providing us the detailed information on the *Fugue* language.

7. REFERENCES

[1] C. Roads, Microsound, MIT Press, 2004.
[2] C. Roads, The Computer Music Tutorial, MIT press, 1996
[3] H. Nishino, LC: A Mostly-strongly-timed Prototype-based Computer Music Programming Language that Integrates Objects and Manipulations for Microsound Synthesis, Ph.D. Thesis, National University of Singapore, 2014
[4] H. Nishino, et al., "The Microsound Synthesis Framework in the LC Computer Music Programming Language." In Computer Music Journal, 39(4), MIT Press, 2016, pp.49-79.
[5] C. Roads, "Introduction to Granular Synthesis," Computer Music Journal 12(2), MIT Press, 1988, pp.11-13
[6] X. Rodet, "Time-Domain Formant-Wave-Function Synthesis," In Spoken Language Generation and Understanding. Springer, pp.429-441.
[7] J.M. Clarke, et al. "VOCEL: New Implementations of the FOF Synthesis Method," In Proc. ICMC, pp.357-371.
[8] T. Wishart, Audible Design. Orpheuse the Pantomime, 1994.
[9] D. Gabor, "Lectures on Communication Theory," Technical Report 238, Massachusetts Institute of Technology, Research Laboratory of Electronics, 1952.
[10] E. Brandt, "Temporal Type Constructors for Computer Music Programming," In Proc. ICMC, 2000.
[11] R. Boulanger. The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming, MIT Press, 2000.
[12] E. Brandt, "Temporal Type Constructors for Computer Music Programming," PhD dissertation, Carnegie Melon University, 2008
[13] A.F. Blackwell and T.R.G. Green, "Notational systems - the cognitive dimensions of notation framework ", In HCI models, theories and frameworks: Toward a multidisciplinary science, Morgan Kaufmann, 2003, pp. 103–134.
[14] S. Wilson., et al., The SuperCollider Book. MIT Press, 2011
[15] S.P. Jones, Haskell 98 Language and Libraries: the Revised Report. Cambridge University Press, 2003.
[16] Y. Minsky, et al., Real World OCaml: Functional Programming for the Masses, O'Reilly, 2013.
[17] M. Odersky, et al., An Overview of the Scala Programming Language., No. LAMP-REPORT-2004-006, École Polytechnique Fédérale de Lausanne, 2004.
[18] W. Apple and P. Jens, Modern Compiler Implementation in Java. MIT Press, 2002.
[19] R. B. Dannenberg., et al., "Fugue: A functional language for sound synthesis," In Computer 24.7, 1991 pp.36-42.
[20] R. B. Dannenberg. In an email exchange, dated Apr 18th, 2016.
[21] D. M. Betz, Xlisp: An Object-Oriented Lisp, Version 2.1., Apple, 1989
[22] R.B. Dannenberg, "The Implementation of Nyquist, A Sound Synthesis Language," In Computer Music Journal 21.3, 1997, pp.71-82.
[23] H. Abelson., and G. J. Sussman, Structure and Interpretation of Computer Programs, MIT Press, 1996.
[24] R. B. Dannenberg, and R. Bencina, "Design patterns for real-time computer music systems," ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music, In Proc. ICMC, 2005.
[25] G. E. Collins, "A method for overlapping and erasure of lists," Communications of ACM, 3 (12), pp.655-657