SPECULATIVE DIGITAL SOUND 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

In this paper, we propose a novel implementation technique, speculative digital sound synthesis, as a practical solution for the tradeoff between computational efficiency and sample-rate accurate control in sound synthesis. Our technique first optimistically assumes that there will be no change to the control parameters for sound synthesis and computes by audio vectors at the beginning of a DSP cycle. Then, after the speculation, when any change is made in the same cycle, it recomputes only the necessary amount of the output.

As changes to control parameters are normally quite sporadic in most situations, recomputation is rarely performed. Hence, the computational efficiency can be maintained mostly equivalent to the computation by audio vectors without any speculation, when no changed is made to the control parameters. Even when any change is made, the overhead can be reduced since the recomputation is only applied to those sound objects that had their control parameters updated, and the output samples in the past are not recomputed.

Thus, our speculative digital sound synthesis technique can provide both sample-rate accurate control in sound synthesis and better performance efficiency by the audio vectors in most practical situations. The tradeoff between these two issues has been a long-standing problem in computer music software design.

1. INTRODUCTION

While the development of faster CPUs in the past decades realized real-time sound synthesis even on laptop computers, the desire for better computational efficiency in digital sound synthesis still remains for many reasons. Computer music practices of our time can involve CPU-intensive sound synthesis techniques. Real-time sound processing of live instruments requires lower audio latency. Audiovisual performances that involves video processing can consume a significant amount of CPU time.

On the other hand, recently there has also been an increasing demand for sample-rate accurate timing behavior recently among computer musicians. For example, it is often required to schedule microsounds with sample-rate accurate timing for microsound synthesis techniques [1]. Inaccuracy in scheduling can lead to a different sound output from that theoretically expected, which can often be audible to human ears. For another example, as Lyon discussed, "a pulsation may feel not quite right when there are a few 20s of milliseconds of inaccuracy in the timing from beat to beat" and "smaller inaccuracy, though rhythmically acceptable, can still cause problems when sequencing sounds with sharp transients, since changes in alignment on the order of a couple of milliseconds will create different comb filtering effects, and the transients slightly realign on successive attacks" [2].

However, there has been a long-standing tradeoff between these two issues, computational efficiency in digital sound synthesis and sample-rate accuracy in timing precision, in computer music software and languages. Generally, the utilization of audio vectors (arrays of sample data) [3, p. 467] is the most popular implementation technique seen in many computer music software and languages to improve computational efficiency in digital sound synthesis. Yet, as output samples within the audio vector are computed at once, the change made to the control parameters can be reflected to the sound synthesis only at the beginning of this computation. Hence, this results in the *control rate* updating the sound synthesis parameters to be lower than the *audio rate* (sample rate) [3, p. 467].

The use of audio vectors, at the cost of sample-rate accurate timing behavior, has been generally considered acceptable in the past, because, given a sufficiently high control rate, human ears are not so sensitive to the slight details in sound output, even when the control rate is set lower than the audio rate. Yet, some of recent computer music programming languages performs sound synthesis using sample-by-sample computation to achieve sample-rate accuracy in timing behavior at the cost of computational efficiency, since there is a significant demand for sample-rate accuracy to support recent computer music practices, as described earlier.

As Moore's law may end [4], it may not be expected that CPUs are significantly faster in the near future. Therefore, it is of significant importance to develop new implementation techniques to achieve computational efficiency even when sample-rate accuracy in timing precision is required. In this paper, we describe a new technique we developed that adapts *speculative computation* [5] to digital sounds synthesis. The technique speculatively computes the audio output using the audio vectors, assuming there is no update to sound synthesis parameters in a DSP cycle, and recomputes only the necessary number of samples for those sound objects that had their sound synthesis parameters updated in the same cycle.

Since, in most practical situations, changes to sound synthesis parameters are made sporadically, and do not occur many times in one DSP cycle, our speculative computation technique can achieve computational efficiency equivalent to the existing techniques to utilize audio vectors when no change is made in a DSP cycle. It also can reduce the overhead for the recomputation, since the recomputation is applied only to the sound objects with updated parameters and the past samples are not recomputed. In other words, our technique provides a practical solution for the problem of the traditional tradeoff between computationally efficiency and sample-rate accurate timing behaivour.

2. RELATED WORK

2.1 Audio Vectors

The simplest approach to implement digital sound synthesis modules is to write a function that computes only a single sample at a time. Figure 1 describes the code example in the C programming language of a table-lookup oscillator that takes this approach, as described by Lazzari in [3, p. 466].

```
A table-lookup oscillator function (1)
float oscil(float  amp,
            float  freq,
            float* table,
            float* index,
            int    len,
            float  sr)
{
  float out;
  out = amp * table[(int) *index];
  *index += freq * len / sr;
  while(index >= len) index -= len;
  while(index < 0) index += len;
  return out;
}

A processing loop to generate a signal
for (int i = 0; i < durs; i++){
  out[i] = oscil(0.5f, 440.f, wtab, &ndx);
}
```

Figure 1. A table-lookup oscillator function that generates a single sample output at each call[1] [3, p. 466].

While the Figure 1 example is quite simple in its implementation, the code, however, is not very computationally efficient, due to the overhead imposed by the function call. Generally, when a function call is made, a computer program must prepare a new *stack frame* (or *activation record*) for a return address, local variables, parameters, and other temporaries [6, Chapter 7], and this imposes the additional overhead that damages computational efficiency in digital sound synthesis. The CPU cache miss may also occur when jumping to the memory space where the code of the function resides. Some may consider that such an overhead can be avoided by using the *macro substitution* [7, p. 89] or *inline function expansion* [8, p. 310]. Yet, such optimization techniques can be applied only at compile time; therefore, it is not applicable to computer music languages, which are normally required to build sound objects (e.g., *instrument* in CSound [9] or *synth* in SuperCollider [10]) dynamically at runtime.

To avoid this overhead by the function call, many computer music programs process audio in blocks by vectors of samples (audio vector). Figure 2 describes an example of this implementation technique as described by Lazzari in [3, p. 467]. As shown, the processing loop is implemented inside the table-lookup oscillator function to process the output samples at once within the function so that the overhead by the function call can be avoided.

In practice, the control parameters for sound synthesis (such as *amp*, *freq*, *table,* and *index* in Figures 1 and 2) and the DSP function often compose a complex data type (e.g., *structure* or *class* in C++) and provides a more general interface (or a type signature of a function/method shared by other sound synthesis modules) so that sound synthesis graphs can be easily constructed at runtime, regardless of the actual type of sound synthesis modules.

```
A table-lookup oscillator function (2)
float oscil(float* output,
            float  amp,
            float  freq,
            float* table,
            float* index,
            int    len,
            int    vecsize,
            long   sr)
{
  // increment
  float incr = freq * length / sr;

  // processing loop.
  for (int i = 0; i < vecsize; i++){
      // truncated lookup
      output[i] = amp * table[(int)(*index)];
      *index += incr;
      while(index >= len) index -= len;
      while(index < 0) index += len;
  }
  return *output;
}
```

Figure 2. A table-lookup oscillator function that utilizes the audio vectors[2] [3, p. 466].

Indeed, the utilization of audio vectors in digital sound synthesis is quite traditional in computer music; *Music V,* developed in 1966 by Mathews et al. [11], is known to be the first language that introduced such "block processing of data among its many extensions" [12]. While the use of audio vectors can significantly improve the computational efficiency, however, there is a side effect that causes a lower control rate than sample rate, as will be discussed in the following subsections.

2.2 Audio Rate and Control Rate

By observing the use of the Music 360 language [13], Vercoe found that "up to 50% of music signal processing is aimed at shaping loudness and pitch contours, – func-

[1] The oscil function has a function prototype with default arguments: *float oscil(float amp, float freq, float* table, float* index, int len-1024, float sr=44100);* [3, p. 466].

[2] Assuming the constants in the code (*def-len,def_vecsize, and def_sr)* are previously defined.

tion essentially of acoustic control that need not be controlled at audio rates" [14]. He developed *Music 11* in 1982, which is known to be the first computer music language in history that introduced the distinction between *audio signal* and *control signal*. As Vercoe discussed, such 'acoustic control' parameters (e.g., envelope shaping, pitch control, vibrate, etc.) can be updated with less expensive data rates without damaging perceptual quality to human ears. Computing such parameters in a lower data rate than sample rate can contribute to significant improvement in computational efficiency. These two different data rates are normally referred to as *audio rate* and *control rate.*

Indeed, the implementation examples in Figures 1 and Figure 2 already illustrate the concept of 'control rate.' In the Figure 1 example, as the oscillator function computers only a single sample at each call, the *freq* and *amp* parameters can be updated when computing each sample. In contrast, in the Figure 2 example, the oscillator function can update these control parameters only before the processing loop computes the output samples. Hence, in the Figure 2 example, the *freq* and *amp* parameters may be seen as control-rate parameters, as they are updated in a lower rate than the audio rate.

Even if we modify the *oscil* function so that it can receive the audio vectors for these two parameters, these audio vectors must be prepared as arguments before each function call. This means these control parameters cannot be updated while the function is processing the audio vectors. Thus, normally, the use of the audio vectors leads to the existence of a control rate that is lower than the audio rate.

2.3 The Issues Regarding the Use of Audio Vectors in Digital Sound Synthesis

In this section, we discuss several issues related to the utilization of audio vectors in computer music systems, to clarify the problem domain that our novel technique of speculation digital sound synthesis handles.

2.3.1 Audio Vectors and Control Rate

As described earlier, while Music 360 already introduced the audio vectors, it still computed all the control parameters for sound synthesis also by the audio vectors. In contrast, Music 11 improved the computational efficiency by introducing *control signals,* in which such 'control-rate' parameters are implemented as scalar values, not as audio vectors[3].

As suggested by this difference between the implementation of Music 360 and Music 11, the existence of the control rate and the use of audio vectors are two separate issues, although there is a strong association between them. For instance, assume a computer music system first updates to all the sound synthesis parameters before processing the sound output, and then performs sample-by-sample computation in the main processing loop until it produces enough number of samples for the current DSP cycle. Such a computer music program has a control rate but does not utilize audio vectors at all. Thus, the association between the use of audio vectors and the existence of the control rate is not an intrinsic issue but is caused by how the digital sound synthesis is implemented.

As will be described in the later sections, our speculative digital sound synthesis technique utilizes the audio vectors but still provides sample-rate accuracy in timing behavior. Regardless of the involvement of audio vectors, the control rate can be equal to the audio rate.

2.3.2 Minimum Feedback Time

Another issue regarding the use of audio vectors is that it imposes a limitation on the minimum feedback time. A sound object composed of sound synthesis modules cannot perform the feedback that is shorter than the size of the audio vectors[4]. Note the issue of minimum feedback time is not directly associated with the control rate. Generally, the control rate is about the minimum interval that can make a valid change to control parameters. This update of the control parameters may be made by control signals (emitted by the control-rate unit-generator) or by the user code external to the sound synthesis graph.[5] In contrast, the limitation on the feedback time is about data-streaming of sample values within a sound object, and there is a significant difference between these two issues, while both can be caused by the use of audio vectors.

Our speculative digital sound synthesis techniques aim at the improvement of the control rate without significant damage to computational efficiency, but does not intend to remove the limitation on the minimum feedback time.

2.4 Speculative Computation

Speculative computation is "an implementation technique that aims at speeding up the execution of programs, by computing pieces of code in advance, possibly in parallel with the rest of the program, without being sure that these computations are actually needed" [6]. The idea of speculative computation (or speculative execution) can be seen in various fields in computer science. For instance, while it is not aiming for the improvement in the performance efficiency, a *backtracking parser*[6] [15], which is a parsing technique invented around 1970, speculatively performs parsing of the input text, and "if the first attempt failed, it rewinds the input and then attempts the next

[3] According to Miller Puckette, in an email exchange dated Apr 6th 2016.

[4] For instance, assume that the size of the audio vectors is two. When computing the first sample in the output audio vector, the next sample to compute with the feedback is also in the same vector, and must be computed before sending the first sample to the feedback loop. Thus, the minimum size of the feedback is two in this case.

[5] One may argue a synthesis patch, as seen in PureData [19], integrates sound synthesis algorithms and control algorithms into one sound object (i.e., 'a patch'). In this case, simply interpret 'sound object' as 'sound synthesis graphs composed of DSP objects'.

[6] A *parser* is a computer program that performs syntactic analysis of the phrase structure of the input program text. It often translates the input program text to abstract syntax trees. The trees are used in the latter phases of compilation.

alternative" [16, p.68]. Such parsing techniques as *memoizing parsing* [17] and *packrat parsing* [18] belong to the same category, but with significant improvement in the performance efficiency in comparison to the original backtrack parsing.

Memory and files are also often predicted and prefetched in many systems to avoid the latency [20][21]. Furthermore, the recent CPUs often predict the execution path and performs speculative execution of code to improve the computational efficiency [22]. Speculation is also applied for parallelization. Thread-level speculation (TLS) "enables the compiler to optimistically create parallel thread despite uncertainty as to whether those threads are actually independent"[7] [23]. Speculative synchronization applies the same concept to barriers, locks, flags, etc. for thread synchronization [24]. Thus, while it may not be directly visible to users, the modern computer system adopts the concept of speculative execution in many important components, mostly to improve the performance efficiency.

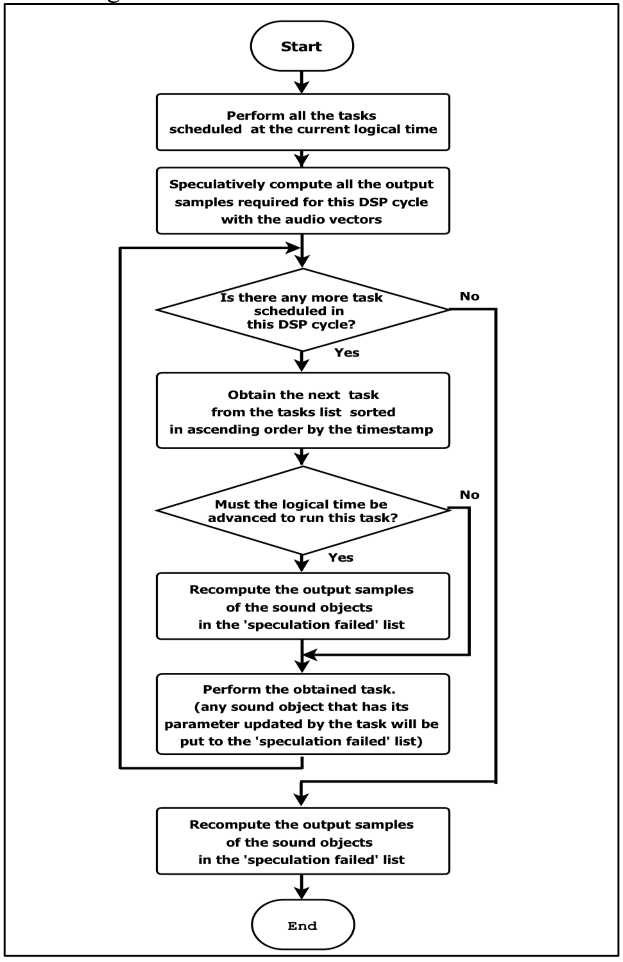
3. DESCRIPTION OF OUR TECHNIQUE

3.1 Overview of Our Technique

Figure 3 illustrates the overall algorithm of our technique to speculatively perform digital sound synthesis. We assume that the computer music system using this technique is based on logical time with sample-rate accuracy, which corresponds to the number of the samples processed since the computer music system began its execution. The system repeatedly executes this algorithm at every DSP cycle.

As shown, this technique first processes all the tasks scheduled right at the current logical time (in the beginning of the DSP cycle). Then, it computes the output samples using the audio vectors with the size required for this DSP cycle at once, optimistically assuming there will be no update to the control parameters in this DSP cycle; thus, the performance efficiency at this phase can be the same as normal digital sound synthesis with audio vectors, as used in many computer music programs.

After this speculative execution phase, the algorithm performs other tasks scheduled in the same DSP cycle one-by-one in ascending order sorted by logical time. If any change has been made to a control parameter that affects the output result in this stage, before advancing the logical time to process the other tasks scheduled in the future (in logical time), the recomputation is performed for those sound synthesis objects with updated parameters. Similarly, in the first phase, it is optimistically assumed that there is no further update to the control parameters in the same DSP cycle, and that the audio vectors are recomputed to the end from the index associated with the current logical time.



Start
Perform all the tasks scheduled at the current logical time
Speculatively compute all the output samples required for this DSP cycle with the audio vectors
Is there any more task scheduled in this DSP cycle?
No
Yes
Obtain the next task from the tasks list sorted in ascending order by the timestamp
Must the logical time be advanced to run this task?
No
Yes
Recompute the output samples of the sound objects in the 'speculation failed' list
Perform the obtained task. (any sound object that has its parameter updated by the task will be put to the 'speculation failed' list)
Recompute the output samples of the sound objects in the 'speculation failed' list
End

Figure 3. The overview of our algorithm to speculatively perform digital sound synthesis.

When all the scheduled tasks are performed in the previous phase, if there still remain any changes made to the control parameters that are not reflected to the output, it recomputes the necessary number of samples, but again, only for those sound objects with updated parameters. The past samples are not recomputed as they are not affected by the change.

Thus, in this algorithm, the tasks are computed with sample-rate accuracy in logical time, and the changes made by these tasks can be reflected to the output with the same timing precision. Hence, while it utilizes the audio vectors, the behavior of the system can achieve sample-rate accurate timing precision. (but note that there is still a restriction on the feedback time due to the involvement of the audio vectors).

Such adaption of speculative execution in digital sound synthesis allows equivalent performance efficiency to normal digital sound synthesis with the audio vectors when the speculation does not fail. It can also achieve less damage to the performance in comparison to sample-by-sample computation in most practical situations, since

[7] For instance, as described by Stefan et al. describes [23], TLS may optimistically parallelize even a 'while' loop that cannot be statically parallelized by the compiler optimization due to the possible data dependences; These threads speculatively execute the loop in parallel, except at least one *safe thread,* which executes code without speculation. If any dependence violation occurs between threads, the threads will redo the computation, except the safe thread(s). Note that the forward progress of execution is guaranteed since there is at least one safe thread.

only those synthesis objects with updated parameters are recomputed, and the past samples, which are not affected by updating, are not recomputed. Normally, it can be expected that most computer music programs just sporadically update sound synthesis parameters and do not update control parameters of many sound objects simultaneously. In the later sections, we discuss a certain situation that may impose a large penalty when speculation fails, and also propose possible solutions.

3.2 Implementation

We implemented our speculative digital sound synthesis technique to investigate whether this technique can provide both computational efficiency and sample-rate accurate timing precision at once, as it is expected in practical situations. Considering the popularity of the unit-generator concept among computer music languages, the sound synthesis framework is designed with unit-generators, while it provides only several simple unit-generators required for the experiments.

To measure the effectiveness of our technique, it is desirable to avoid any influence unrelated to the computational efficiency in digital sound synthesis, such as the overhead imposed by garbage collections, memory allocation, and the control tasks. For this reason, we implemented a simple sound synthesis framework from scratch in C++, a programming language without the automatic memory management feature so that we can measure only the computational efficiency in digital sound synthesis.

```
The SpecUGen class
01:class SpecUGen
02:{
03:public:
04:  SpecUGen(SpecPatch* patch, int64_t avecSize);
05:  virtual ~SampUGen(void)
06:
07:  virtual lc_sample* compute()=0;
08:  virtual lc_sample* recompute(int64_t offset)=0;
09:  virtual lc_sample* getOutput()=0;
10:
11:  virtual int64_t getAvecSize();
12:
13:protected:
14:  SpecPatch* patch;
15:  int64_t    avesSize;
16:};
```

Figure 4. The declaration of the *SpecUGen* abstract class.

```
The SpecPatch class
01:class SpecPatch
02:{
03:public:
04:  SpecPatch(LCAudioEngine* ae, int64_t avecSize);
05:  virtual ~SpecPatch(void)
06:
07:  virtual void compute();
08:  virtual void recompute(int64_t offset);
09:  virtual void setOutputUGen(SpecUGen* ugen);
10:
11:  virtual lc_sample* getOutput (void);
12:  virtual int64_t    getAvcSize(void);
13:
14: protected:
15:  int64_t        avecSize;
16:  SpecUGen*      outputUGen;
17:  LCAudioEngine* audioEngine;
18:};
```

Figure 5. The declaration of the *SpecPatch* class.

Figures 4 and 5 show the declaration of the base abstract class for all unit-generators (*SpecUGen*) and the class for sound objects (*SpecPatch*), respectively. The *SpecPatch* object is a sound object that is responsible for providing the output samples to the sound synthesis engine in each DSP cycle. As the software framework for this experiment is designed just to measure the actual performance efficiency of our technique, the design is quite simple and only pulls the output samples from the unit-generator object given by the *setOutputUGen* method call.

```
01:lc_sample* SCSineOsc::compute(void)
02:{
03:  //adjust phase for the better precision.
04:  phase = fmod(phase, 2 * LC_PI);
05:
06:  //perform speculative computation.
07:  for (int64_t i = 0; i < this->avecSize; i++){
08:    output[i]    = (lc_sample)sin(phase) * amp;
09:    pastPhase[i] = phase;
10:    phase += phaseInc;
11:  }
12:  return output;
13:}
14:
15:lc_sample* SCSineOsc::recompute(int64_t offset)
16:{
17:  //adjust phase for the better precision.
18:  phase = fmod(pastPhase[offset], 2 * LC_PI);
19:
20:  //perform recomputation.
21:  for (int64_t i = offset; i < avecSize; i++){
22:    output[i] = (lc_sample)sin(phase) * amp;
23:    pastPhase[i] = phase;
24:    phase += phaseInc;
25:  }
26:  return output;
27:}
28:
29:void SCSineOsc::setFreq(LCAudioEngine* engine,
30:                        double         freq  )
31:{
32:  //compute the new phase increment.
33:  phaseInc = 2 * LC_PI * freq / GetSampleRate();
34:  this->freq = freq;
35:  engine->notifyUpdate(this->patch);
36:
37:  return;
38:}
```

Figure 6. The implementation of the *compute, recompute, and setFreq* methods in the *SUSineOsc* class.

The *compute* method declared in Figure 5 (line 07) is used for speculative computation, which is called right after all the tasks scheduled at the beginning of the DSP cycle were executed, and the *recomputed* method (line 08) is called to recompute the output when any change is made to the control parameters of the unit-generators in the sound synthesis graph of this patch object, receiving an offset in the samples from the beginning of the current DSP cycle (the *offset* argument). The methods in the *SpecUGen* class are designed quite similarly. The *compute* method in Figure 4 (line 07) is called during the speculative computation in the beginning of DSP cycle, traversing the sound synthesis graph. The *recomputed* method is called during the recomputation.

Figure 6 illustrates the actual implementation of these methods in the *SUSineOsc2* class. As shown, it stores the phase information in the internal buffer (*pastPhase*) during the DSP cycle so that it can recover the related parameters at the point where the recomputation must begin. Note that some unit generators may not have to store the past parameters. For example, since the white noise unit-generator (*SUWhieNoise*) does not depend its output on any previous sample at all, it is not necessary to store the past information. It should also be noted that the update of the control parameter is reported to the sound synthesis engine in the *setFreq* method (line 35) so that the patch can be added to the 'speculation failed' list.

As for the creation of a new sound object (*SpecPatch)* during the DSP cycle (not at the beginning of the cycle), it is normally required just to put the new sound object to the 'speculation failed' list. By initializing the internal buffers with zero and computing from the given offset, the *recompute* method can be called to produce the output at the timing of its instantiation and the output before the instantiaion can be zero-cleared. It should be noted that it is often even unnecessary to let the unit-generator objects handle such instantiation during the DSP cycle. For example, in the Figure 6 example, the *recompute* method of the *SCSineOsc* class does not require any special treatment, and the computation of the sine wave output can start from the given offset without any problem. Even if there is any special care for the instantiation during the DSP cycle, only one flag variable is required. If the *recompute* method is called before the *compute* method, the sound object was instantiated after the speculative computation was already performed, which means it was instantiated during the DSP cycle.

4. PERFORMANCE MEASUREMENT

4.1 The Test Environment

Since the software framework is designed just for the measurement of the actual performance efficiency of our speculative digital sound synthesis technique, it provides only several simple unit-generators, as shown in Table 1. However, these are enough to perform the test tasks described in Table 2. To compare with the existing technique, we also implemented the equivalent versions to perform sound synthesis sample-by-sample within the same framework and the normal audio-vector-based sound synthesis without speculation.

SUSig: converts a float value to an audio signal.
SUWhiteNoise: generates a white noise signal.
SUSineOsc: generates a sine wave signal. The frequency and amplitude parameters are specified by floating point values.
SUSineOsc2: generates a sine wave signal. The frequency and amplitude parameters are given as input audio signals.
SUArithmeticOp: performs the arithmetic operations (+, -, *, /) to two input signals.
SUEnvelope: generates an envelope signal

Table 1. The list of the available unit-generators

We performed experiments to compare the performance efficiency of our speculative digital sound synthesis technique, sample-by-sample sound synthesis, and popular audio-vector-based sound synthesis without speculation. We excluded other factors as much as possible, such as memory allocation, and execution of scheduled tasks for sound synthesis control, and measured only the CPU time consumed by the part of the code where digital sound synthesis is performed. 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 complied with the '*-Ofast*' option (the fastest-aggressive optimization) with the Apple LLVM7.1 compiler for all the test tasks.

4.2 The Test Tasks

Table 2 shows the test tasks prepared for the performance measurement. These test tasks are designed under the assumption that the update to control parameters from the program external to the sound object is sporadic, as it is in most practical situations. Each task has two sub tasks, and is performed for three different implementations: (a) our speculative digital sound synthesis technique, (b) normally block processing by audio vectors without speculation, and (c) sample-by-sample processing. We used C++'s *std::mt19937*, a *Mersenne twister* random value generator class, with the same seed value. Hence, the events can be generated with the same timestamps and the same parameters for all these implementations for (a), (b) and (c).

Task #1: Sine Wave Oscillators Ten sine wave oscillators are created.
Task #2: Additive Synthesis Ten additive synthesis instruments are created. Each of them consists of four sine wave oscillators and one envelope applied to the entire output.
Task #3: FM Synthesis Ten simple FM synthesis instruments are created. The unit-generator graph of this FM synthesis instrument is shown in Figure 7.
****Each of the above tasks has these two subtasks***
Sub task #1: Each instrument will update the base frequency (Task #1-#2) or the carrier and modular frequency (Task#3) to a random frequency (or random frequencies) with the interval of 50 msec, one-by-one in turn (only one instrument updates its frequency at the same time).
Sub task #2: One of the instruments is randomly picked up and will update the base frequency (Task #1-#2) or the carrier and modular frequency (Task#3), with the random intervals between 10 msec to 100 msec.

Table 2. The test tasks for the performance measurement

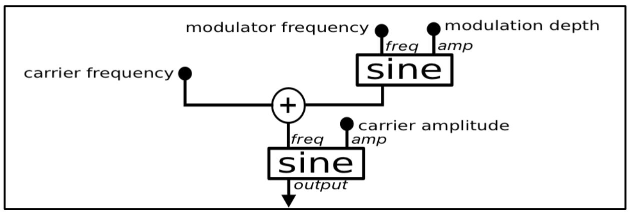


Figure 7. A simple FM synthesis instrument

To measure the exact CPU time without any influence of the task switching to other processes by the operating system, we used the *clock* C library function, which can measure only the CPU time used by the DSP process. The CPU time is measured only for the code related to the digital sound synthesis, excluding other parts, such as initialization, memory allocation, and the scheduled control tasks. Both the size of audio vectors and the DAC I/O block size are set to 256 samples. Each task continues for ten seconds and is repeated five times, to obtain the CPU time used for both the worst case and the best case (or the max and min CPU time) among all the DSP cycles, and the average of the CPU time over all the DSP cycles.

4.3 Test Results

Table 3 shows the test results for the three tasks. The column '*recom. overhead*' shows the overhead imposed when the speculation fails (*'the CPU time for recomputation'* / '*the overall CPU time in each DSP cycle'*).

Algorithm	*avg. (msec)*	*max (msec)*	*min (msec)*	*recom. overhead*	
				avg (%)	*max(%)*
Task 1: Sine Wave Oscillators					
Sub Task 1					
sample-by-sample	0.250	1.058	0.125		
audio vector	0.128	0.575	0.061		
speculative	0.127	0.512	0.060	4.5%	22.2%
Sub Task 2					
sample-by-sample	0.224	0.719	0.126		
audio vector	0.115	0.825	0.064		
speculative	0.124	0.490	0.061	4.2%	13.9%
Task 2: Additive Synthesis					
Sub Task 1					
sample-by-sample	0.679	1.853	0.355		
audio vector	0.357	1.293	0.175		
speculative	0.345	1.174	0.177	5.0%	9.3%
Sub Task 2					
sample-by-sample	0.636	2.561	0.354		
audio vector	0.339	1.291	0.179		
speculative	0.328	1.046	0.180	4.6%	11.6%
Task 3: FM Synthesis					
Sub Task 1					
sample-by-sample	0.392	1.115	0.180		
audio vector	0.200	0.754	0.084		
speculative	0.200	0.654	0.080	5.3%	22.1%
Sub Task 2					
sample-by-sample	0.258	0.711	0.182		
audio vector	0.194	0.753	0.082		
speculative	0.189	0.798	0.084	5.7%	19.6%

Table 3. The test results.

5. DISCUSSION

5.1 The Evaluation of the Test Results

As shown in Table 3, the use of audio vectors significantly improves the performance efficiency (about twice as fast as the sample-by-sample computation). Yet, surprisingly, no significant decrease in the performance was observed in speculative implementation in comparison to the traditional audio vector implementation, while allowing sample-rate-accurate updates of the control parameters. While the overhead imposed by the recomputation is surely observed, all of the worst case (max) time, the best case time (min), and the average CPU time are almost equivalent and the difference is mostly negligible (sometimes a bit faster but still negligible) for all the test tasks. The recomputation overhead is about 5% on average and about 20% in the worst-case in all these tasks; yet, this overhead seems to not cause any damage to the overall performance efficiency in the test tasks.

Such a result is quite favorable. Our technique is likely to achieve a performance efficiency almost equivalent to the traditional audio vector implementation, since what matters for the performance efficiency in real-time sound synthesis is the worst-case execution time (to meet the sound output deadline) and the average CPU time (to give more CPU time to other processes to perform their tasks). One reason for this equivalent performance efficiency in the average CPU time may be due to our assumption that the updates to the control parameters are fairly sporadic and that the test tasks are designed upon this assumption. If the recomputation is not frequently performed, it may not affect the overall average CPU time much.

While there is certainly the necessity for a more detailed investigation, another plausible hypothesis behind the equivalent performance in the worst-case CPU time would be that the time cost imposed by the CPU cache misses may have overshadowed the cost of recomputation. While we measured only the CPU time used by the sound synthesis program using the *clock* library function, as the utilization of the cache memory by other processes can lead to unpredictable cache misses during sound synthesis, the time costs of which are imposed to the sound synthesis program, and can be quite large in comparison to the time costs for a simple DSP algorithm. This can occur anytime also in a practical situation during a live computer-music performance.

5.2 Reducing the Penalty of Speculation Failure

Our technique is based on the assumption that the update to the control parameters is fairly sporadic. Hence, if there are too many updates performed in one DSP cycle, the overhead caused by the recomputation can be unneglectable. One way to reduce this overhead is to recompute only the part of the sound synthesis graph, only where the update actually influences the output. This can be done by *data dependence analysis* [25] in the sound synthesis graph.

Another way to reduce this overhead is to divide a DSP cycle into several blocks. For instance, if the outputs of 256 samples are required for one DSP cycle, instead of making the size of audio vectors to 256, one can make it 32 and performs the speculative computation eight times. In this case, the number of the samples to recompute when speculation fails will reduce to 31 even in the worst case (about 12% of 255, when the whole output is recomputed when the audio vector size is set to 256), if there is no further update in the performed in the same DSP cycle. Further reduction of the overhead may be achieved by prediction. Generally, speculative execution often utilizes prediction to improve the performance efficiency [26]. By adaptably changing the number of samples to speculatively compute each sound object based on the past speculation results, the penalty caused by the speculation failure can be reduced.

5.3 Realizing Single Sample Feedback

As discussed earlier, the issues of control rate and minimum feedback time deal with different problem domains and are out of the scope of this paper. However, we suggest that just-in-time compilation [27] would be beneficial to solve this problem by compiling the while unit-generator graph into the native code.

5.4 Multithreading

Our technique can be extended to compute the future output (e.g., for the next two seconds), and this extension can be performed by other threads independently from the main audio computation thread. When the speculation fails, the recomputation is performed first only for the current DSP cycle by the audio thread and then other threads can continue speculation. Such extension can contribute to the further improvement of the performance efficiency, especially when used with multicore CPUs.

6. CONCLUSIONS AND FUTURE WORK

We described our novel technique that speculatively performs digital sound synthesis with audio vectors and recomputes only when any update to the control parameters is made. This technique contributes to achieving both the performance efficiency and sample-rate accurate control at once in most practical situations. As described in the discussion, we observed almost no damage to our simple test tasks.

Since this tradeoff between the performance efficiency and sample-rate accurate control of sound synthesis parameters has been one of the long-standing problems in computer music software, such a speculative digital synthesis technique can be quite beneficial for next-generation computer music systems to provide both high performance real-time sound synthesis and sample-rate accurate timing behavior at once, since a significant improvement of the clock speed of CPUs can be hardly expected in the near future.

For future work, we are planning to investigate the performance efficiency in more complicated sound synthesis techniques. We also plan further extension of our speculative digital sound synthesis technique, as we discussed in the *Discussion* section, together with the implementation of this technique into the LC computer music language that we are currently developing [28].

Acknowledgments
We would like to express our gratitude to Prof. Miller Puckette for the information on the implementations of Music 360 and Music 11.

7. REFERENCES

[1] C. Roads, *Microsound*, MIT Press, 2004.
[2] E. Lyon, "A Sample Accurate Triggering System for Pd and MaxMSP," *Proc. ICMC,* 2006.
[3] R. Boulanger et al., *The Audio Programming Book*, MIT Press, 2010.
[4] M. Waldrop, "The Chips are down for Moore's law," *Nature News,* 2016.
[5] W. Apple and P. Jens, Modern Compiler Implementation in Java. MIT Press, 2002.
[6] G. Boudl, and G. Petri, "A theory of speculative computation," *Programming Languages and Systems.* Springer Berin Heidelberg, 2010, pp. 165-184.
[7] B.W. Kernighan and D.M. Ritchie, The C Programming Language (2nd edition), Prentice Hall, 1988.
[8] B. Stroustrup, The C++ programing language (4th edition), Addison-Wesley Professional, 2013.
[9] R. Boulanger et al., *The Csound book,* MIT Press, 2000.
[10] S. Wilson et al., *The Supercollider Book*, The MIT Press, 2011.
[11] M. V.Mathews, et al., *The technology of computer music*, MIT Press, 1969.
[12] B. Vercoe, "New Dimensions in Computer Music," *Trends & Perspective in Signal Processing, 2(2),* 1982.
[13] B. Vercoe, "The MUSIC 360 language for digital sound synthesis." *Proc. of the American Society of University Composers",* 6, 1971.
[14] B. Vercoe, "Computer systems and languages for audio research," *Proc. Audio Engineering Society Conference*, Audio Engineering Society, 1982.
[15] A. Birman and J. D., Ullman, "Parsing algorithm with backtrack," *Information and Control, 23(1),* 1973.
[16] T. Parr, Language Implementation Patterns, Pragmatic Bookshelf, 2009.
[17] P. Norvig, "Techniques for automatic memoization with applications to context-free parsing," *Comput. Linguist., 17(1),* 1991.
[18] B. Ford, "Packrat parsing:: simple, powerful, lazy, linear time, functional pearl," *Proc. of ACM SIGPLAN*, 2002.
[19] M. Puckette, "Pure Data: another integrated computer music environment." *Proc. ICMC,* 1996.
[20] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," *Proc. USENIX summer 1994 technical conference,* 1994.
[21] H. Zhigang et al., "Timekeeping in the memory system: Predicting and optimizing memory behavior", *Proc. Computer Architecture,* 2002.
[22] A. S. Tanenbaum and T. Austion, "4.5.4 Speculative Execution", *Structured Computer Organization (The Sixth Edition)*. Pearson, 2012.
[23] J. G. Steffan, et al., "A scalable approach to thread-level speculation,", *Proc. ISCA*, 2000.
[24] J.F. Martinez and J. Torrelas. "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," *ACM SIGOPS Operating System Review*, *36(5)*, 2002.
[25] D. E. Maydan et al., "Efficient and exact data dependence analysis," *ACM SIGPLAN Notices, 26(6),* 1991
[26] A. Mendelson and F. Gabbay, "Speculative execution based on value prediction," *Technical report, Technion-Israle Institute of Technology,* 1997.
[27] J. Aycock. "A brief history of just-in-time." ACM Computing Surveys, Vol.35(2), 2003.
[28] H, Nisino et al. "LC: A New Computer Music Programming Language with Three Core Features", *Proc. ICMC-SMC 2014*