Unit-generator Graph as a Generator of Lazily Evaluated Audio-vector Trees

Hiroki Nishino
Department of Industrial Design,
Chang Gung University, Taiwan
hiroki.nishino@acm.org

ABSTRACT

Computer music systems normally implement a unit-generator graph as a network of unit generators, through which audio vectors are streamed and processed. This paper proposes an alternative implementation technique for unit-generator-based sound synthesis, which views a unit-generator graph as a generator of audio-vector trees to be lazily evaluated. The simplest implementation of this technique allows to process sound synthesis and sound-control tasks in different threads even in a synchronous computer music system, making real-time sound synthesis more stable by amortizing the time costs for sound-control tasks over DSP cycles, while maintaining the low roundtrip latency between the audio input and processed output at the same time. We also extend the discussion to the possible extensions of our technique for parallelization, distribution, and speculation in real-time sound synthesis. The investigation into such a novel implementation technique would benefit further research on high-performance real-time sound synthesis.

1. INTRODUCTION

When the unit-generator concept is implemented, the most common practice is to implement it as a network of unit generators, through which samples are streamed and processed. While such an implementation is considered acceptable, there exist some issues to consider in synchronous computer music systems. Synchronous computer music systems must not advance the logical synchronous time to compute output samples from unit generators until all the scheduled tasks at the timing are completed. This behavior allows precise timing control in a computer music program, which is quite favorable in many aspects. However, as sound-control tasks must be performed normally within the same thread that computes output samples to realize such synchronous behavior, synchronous computer music systems must deal with the tradeoff between the stability in real-time sound synthesis (by amortizing the time cost for sound control tasks over DSP cycles) and the latencies (in the audio I/O and interaction).

We propose an alternative implementation technique, in which a unit-generator graph is modelled as a generator of trees of audio vectors. These audio-vector trees, each node of which contains an audio vector and sound synthesis parameters for unit generators, are lazily evaluated when the output samples are required. While our technique adds some extra overhead for the generation and release of audio-vector trees, it allows us to divide real-time sound synthesis and sound control tasks into separate threads without damaging the precise synchronous behavior. Moreover, the technique can improve the stability in real-time sound synthesis by amortizing the time cost for sound-control tasks over DSP cycles, without increasing the roundtrip latency between the audio input and processed output. The action-to-sound latency still exists in our technique, yet human perception is much more generous to the action-to-sound latency than the roundtrip latency.

In addition to such favorable characteristics, our technique also has a good potential to be further extended for parallelized and distributed sound-processing since the sound synthesis parameters within audio-vector trees are designed to not be updated after the generation and to return identical output whenever evaluated. Hence, it is possible to parallelly evaluate audio-vector trees by multi-threading without caring about the data race or distributing audio-vector trees, which may be still unevaluated or just partially evaluated, to other computers in the network. Our technique may also be potentially beneficial for speculative digital sound synthesis [1] to effectively perform speculation for future output samples, as the evaluation is performed lazily. As Moore's law may end [2] and the advance of CPU speed cannot be expected to continue as it has in the past, it is desirable to investigate such an implementation technique with potentially benefits for high-performance real-time sound synthesis.

2. RELATED WORK

2.1 Real-time Audio Programming

2.1.1 Audio thread

Real-time computer music programs must perform two main tasks. One is sound synthesis and the other is compositional algorithm/sound control. For the task of sound synthesis, many operating systems provide APIs for a real-time audio application, which invoke a callback function or notify a thread when the next chunk of output samples

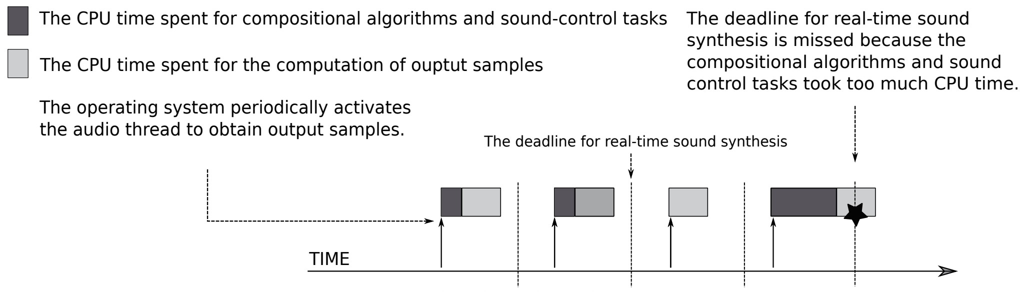
The CPU time spent for compositional algorithms and sound-control tasks
The CPU time spent for the computation of ouptut samples
The operating system periodically activates the audio thread to obtain output samples.
The deadline for real-time sound synthesis
The deadline for real-time sound synthesis is missed because the compositional algorithms and sound control tasks took too much CPU time.
TIME

Figure 1. Audio thread implementation.

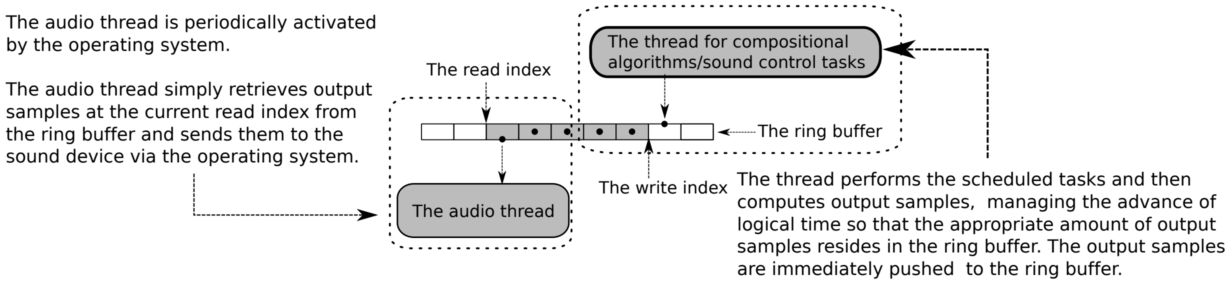
The audio thread is periodically activated by the operating system.
The audio thread simply retrieves output samples at the current read index from the ring buffer and sends them to the sound device via the operating system.
The read index
The thread for compositional algorithms/sound control tasks
The ring buffer
The write index
The audio thread
The thread performs the scheduled tasks and then computes output samples, managing the advance of logical time so that the appropriate amount of output samples resides in the ring buffer. The output samples are immediately pushed to the ring buffer.

Figure 2. Ring buffer implementation.

is required for real-time sound output. Such a callback/notification is performed asynchronously by the operating system without coordinating the invocation/notification with the user program. The list of such APIS include Core Audio (Mac OS X) [3], ALSA (Linux) [4, p.22], WASAPI (Windows) [5], and ASIO (multiplatform) [6][1]. As to avoid unnecessary complication in the discussion, we simply utilize the term: 'the audio thread' to represent the thread that provides the output samples for the sound output (via the operating system so that it can send the output samples to the sound device), regardless of whether is the thread created by the operating system to invoke the callback function or if it is the thread created by a user program to wait for a notification. In either case, the thread is expected to provide output samples for the next DSP cycle as fast as possible so that it can safely meet the deadline for real-time sound synthesis.

Note that 'the audio thread' does not refer to where the output samples are computed. It simply refers to the thread that provides the output samples for the sound device via the operating system, regardless of whether the output samples are computed within the thread. We also represent both compositional algorithms and sound-control tasks by the same term: 'control tasks' to avoid redundant expressions in the discussion.

2.1.2 Synchronous Programming

The synchronous programming concept is based on the ideal synchronous hypothesis, which assumes that "ideal systems produce their outputs synchronously with their inputs" and that "all computation and communication are assumed to take zero time (that is, all temporal scopes are executed instantaneously)" [7, p.360]. This concept is clearly unrealizable as it is, as it requires an infinitely fast computer. Hence, the ideal synchronous hypothesis is interpreted to imply "the system must execute fast enough for the effects of the synchronous hypothesis to hold," when implementing a synchronous system [7, p.360].

The concept of synchronous programming is quite beneficial for a computer music system to perform scheduled control tasks conceptually at the exact timing without the advance of real-time sound synthesis. In practice, real-time computer music systems with synchronous behaviors normally interpret the ideal synchronous hypothesis to imply as follows:

1) The system utilizes the logical synchronous time.
2) The logical synchronous time can't be advanced until all the tasks scheduled at a certain timing are completed.
3) When the logical synchronous time is advanced, then the output samples are computed (e.g., the unit generators generate their outputs).
4) The advance of the logical synchronous time must be coordinated with the advance of physical time so that it can provide output samples as required for the real-time sound output.
5) Sound synthesis and control tasks must be finished fast enough to meet the deadline for real-time sound synthesis.

When implementing a synchronous computer music system, sound synthesis and control tasks are normally computed within the same thread if no special implementation technique is involved. Most commercial operating systems cannot perform a context switch to coordinate two threads fast enough even at the control rate of a computer music system. Hence, if control tasks and sound synthesis are performed in different threads, it is virtually impossible to realize synchronous behavior of a computer music system in real-time; yet, if both sound synthesis and control tasks

[1] Some APIs also allow a user program to actively query if there is the necessity to provide new output samples. For instance, ALSA provides such APIs. In this case, the implementation will be similar to the ring buffer implementation described in the section *2.1.4. Ring buffer implementation.*

are performed in the same thread so that synchronous behaviors of a computer music system can be realized, it also leads to some issues to consider, as described in the following sections.

2.1.3 Simple audio thread implementation

A simple implementation technique for unit-generator-based sound synthesis to realize synchronous behavior is to perform the computation of output samples from unit-generator graphs within the audio thread. This implementation technique contributes to reduce 'the roundtrip latency' (the latency between the audio input and the processed output), since the most recent input is passed to the audio thread when it is activated, and unit-generators can utilize the input samples immediately when computing the requested output samples.

However, computer music systems must also process control tasks. If they are also processed within the same thread where sound synthesis is performed, it also influences the overall performance efficiency of real-time sound synthesis since the control tasks also impose some CPU time cost. In the worst case, the deadline for read-time sound synthesis can be missed if there is any task that consumes too much CPU time to be completed. Figure 1 visually illustrates such a situation.

2.1.4 Ring buffer implementation

Another typical implementation technique for a synchronous computer music system is to compute output samples in different thread than the audio thread. The thread pushes output samples into a ring buffer, from which the audio thread retrieves the next output samples. Figure 2 illustrates such an implementation technique. In an implementation of this kind, it must be guaranteed that there are enough output samples in the ring buffer when the audio thread is activated asynchronously by the operating system. If there are not enough samples, that means the deadline for real-time sound synthesis is missed. To ensure that there are always enough samples in the ring buffer, the ring buffer implementation must compute output samples ahead of time, observing how many output samples are currently available in the ring buffer.

Moreover, the input samples must be pushed to another ring buffer by the audio thread so that samples can be utilized for sound processing in the ring buffer implementation. This buffering can increase the roundtrip latency, as the most recent input samples cannot be utilized when computing the output samples. The buffering can also lead to the increase in the action-to-sound latency in reacting to external events (such as MIDI input), which may be perceptible if the number of pre-generated output samples that reside in the ring buffer is large[2]. Hence, it is desirable to keep the amount of the output samples computed ahead of time as small as possible to minimize the roundtrip latency and the action-to-sound latency, while ensuring that it is also large enough so that the audio thread can always obtain sufficient samples

However, the ring buffer implementation has a benefit that it is not bounded by the deadline for real-time sound synthesis within the audio thread. Even when a certain control task consumes the CPU time more than the duration of one DSP cycle, it may be possible to safely perform real-time sound synthesis, since the time cost for the control task can be amortized among DSP cycles. This can improve the stability in real-time sound synthesis.

For instance, assume the max CPU time available for one DSP cycle is 10ms, and sound synthesis for one DSP cycle requires 6ms. Even when a certain task requires 5ms, it is possible to meet the deadline if the cost can be amortized over DSP cycles. If the cost is amortized over two cycles, the cost will be 6ms * 2 + 5ms = 17ms, which is less than the 10ms * 2 = 20ms deadline for two DSP cycles. If the control tasks and sound synthesis can be finished before the audio thread consumes all the output samples in the ring buffer, there will be no problem in real-time sound synthesis. Thus, the ring buffer implementation can improve the performance efficiency of the entire computer music program (sound synthesis + control tasks) and can also lead to more stable behavior in real-time sound synthesis, yet at the cost of the roundtrip latency and action-to-sound latency.

2.1.5 Tradeoff between roundtrip/action-to-sound latencies and stability in sound synthesis.

As discussed, the simple implementation to perform both sound synthesis and control tasks in the audio thread achieves better roundtrip and action-to-sound latencies than the ring buffer implementation. Instead, the ring buffer implementation can achieve the amortization of the time cost for control tasks, which can lead to the overall improvement in the performance efficiency and more stable behavior. In the case in which the realization of synchronous behavior is important for a computer music system, these two simple but traditional implementations cannot avoid this tradeoff as control tasks and sound synthesis must be performed within the same thread.

2.2 Update-caching Technique

The update-caching technique [10], which we previously developed, can also be utilized to solve this tradeoff between the roundtrip/ action-to-sound latencies and stability in real-time sound synthesis. In the update-caching technique, update events of sound synthesis parameters are not instantly applied. Instead, they are cached with timestamps in logical time within unit-generators. When sound synthesis is performed, the cached events are applied within the main loop to compute output samples of a unit-generator. Hence, the update-caching technique can realize the sample-rate accurate control of sound synthesis parameters. Our preliminary study suggests that the performance

[2] While professional pianists may notice the latency even under 10 ms [8], the just noticeable difference (JND) for gestural sound control without tactile feedback is estimated between 20 and 30 msec [9].

efficiency is almost the same as the simple audio-vector implementation (without the sample-rate accurate behavior) when the update is not frequent and is also almost equivalent to the sample-by-sample computation implementation when the update events are very frequent (e.g. in every two samples). Thus, the update-caching technique can be utilized to solve the tradeoff between the roundtrip latency and the stability in sound synthesis, while the action-to-sound latency still exists. Even when the computation of output samples and the execution of control tasks are performed in separate threads, synchronous behavior can be achieved since the updates are cached with logical-time stamp and applied to the sound synthesis right at the timing in logical time,

2.3 Lazy Evaluation in Sound Synthesis

There are not many previous works that utilize lazy evaluation in sound synthesis. Yet, there are only a few examples in non-real-time sound synthesis languages. The earliest known example is the *Fugue* computer music language [11]. In Fugue, output samples are not computed at the timing when instruments and scores are built, and the computations occur when the score is evaluated for playback. However, unlike many other languages with lazy evaluation, Fugue doesn't memoize the result of intermediate computation by default [12]. In Fugue, the lazy evaluation is applied to eliminate unnecessary memory allocation and signal copying to form intermediate results so that the performance efficiency can be improved. Fugue was later developed into Nyquist [13]. Nyquist also perform lazy evaluation in sound synthesis, yet it utilizes block processing (by audio vectors) so that the required memory spaces for sound synthesis can be significantly reduced, while Fugue was still implemented to allocate sufficient memory for the entire results and computes output samples one at a time. The intermediate results were also not memoized by default in Nyquist [12]. Chronic is a recent example of a non-real-time computer music language that utilizes lazy evaluation in sound synthesis [14]. Chronic adopts lazy evaluation so that it can express an audio stream of infinite length, without involving data-streaming objects (such as unit-generators).

An example of utilizing lazy evaluation in real-time sound synthesis is rare. We previously utilized lazy evaluation for real-time sound synthesis in [12], yet it is applied to distribute the time cost for the evaluation of the 'microsound object' (which is essentially an audio vector of arbitrary length with various utility methods) over DSP cycles so that the system can evaluate large objects, the sizes of which are far beyond the duration of microsounds (~ 100ms [15, p.21]), without the temporal suspension of real-timed sound synthesis. The motivation behind this work is to extend the application domain of our sound synthesis framework as seen in the LC computer music language [16] for more general sound synthesis techniques beyond microsound synthesis; yet, this technique is not applicable to unit-generator-based sound synthesis due to the significant difference in abstractions of sound synthesis.

3. DESCRIPTION OF OUR TECHNIQUE

3.1 Overview of Our Implementation Technique

Unlike implementations in existing computer music systems, our technique views a unit-generator graph as a generator of tress of audio-vector nodes, each of which is coupled with sound synthesis parameters. The computation of output samples is lazily performed when output samples are requested for the sound output.

Our technique expects to compute output samples within the audio thread and to utilize another thread for control tasks. The computation of output samples is divided into three phases. In the first phase, the generation of audio vector tress is performed within the thread for control tasks (not in the audio thread). When the logical time is advanced, the control thread traverses unit-generator graphs and generates trees of audio-vector nodes (Figure 3 left, in the next page). This advance is performed synchronously. In other words, the logical time is advanced only after all the control tasks scheduled at the timing are completed. In the second phase, the trees are pushed into a FIFO queue, from which the audio thread can retrieve them when new output samples are requested. When the audio thread is activated, it evaluates the audio vector trees (Figure 3 right) to obtain the output samples for the DSP cycle. In the last phase, the audio vector trees are released (Figure 3 below).

3.2 Generating Audio-vector Trees

In the generation phase of audio-vector trees, we utilize the object pool pattern [17] so that the time cost for memory allocation can be avoided. When a unit generator graph is traversed in depth-first order, each unit-generator takes out an audio-vector node from its own object pool. The audio-vector node is a data structure that couples an audio vector and sound synthesis parameters, and these parameters must be sufficient to compute output samples for the DSP cycle without depending any other external parameter. The pointer to the unit generator is also included in these parameters. This pointer is later used to call back the method to compute output samples, to which the coupled sound synthesis parameters are given as arguments.

After setting up an audio-vector node, the unit generator updates its internal sound synthesis parameters for the next DSP cycle. Note that the next internal parameters of each unit generator can be often obtained without processing any output sample. For example, the phase of a sinewave at the beginning of the next cycle can be directly obtained by a simple expression: "phase_increment_per_sample * audio_vector_size + current_phase" (if the update is performed only at the control rate). Hence, it is expected that the time cost for the tree generation is not large.

There are some cases that require more consideration. For example, there are many unit generators that require the past samples; FIR and IIR filters are typical examples of this kind. For another example, even a sinewave oscillator may require such a consideration when its frequency can be updated at the audio rate. In this case, the last phase

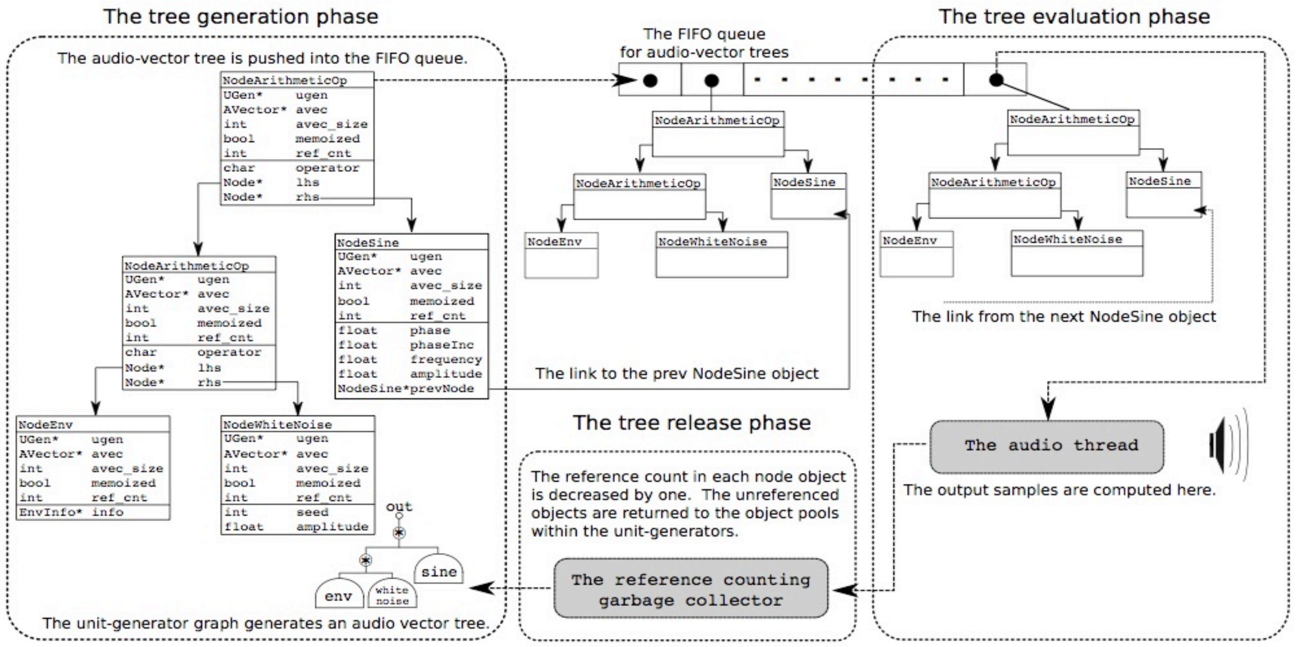
The tree generation phase
The audio-vector tree is pushed into the FIFO queue.
NodeArithmeticOp
UGen* ugen
AVector* avec
int avec_size
bool memoized
int ref_cnt
char operator
Node* lhs
Node* rhs
NodeSine
UGen* ugen
AVector* avec
int avec_size
bool memoized
int ref_cnt
float phase
float phaseInc
float frequency
float amplitude
NodeSine* prevNode
NodeEnv
EnvInfo* info
NodeWhiteNoise
int seed
float amplitude
out
env
white noise
sine
The unit-generator graph generates an audio vector tree.
The FIFO queue for audio-vector trees
The link to the prev NodeSine object
The tree evaluation phase
The link from the next NodeSine object
The audio thread
The output samples are computed here.
The tree release phase
The reference count in each node object is decreased by one. The unreferenced objects are returned to the object pools within the unit-generators.
The reference counting garbage collector

Figure 3. The overview of our implementation technique.

value in the previous DSP cycle is required to compute the output samples in the current cycle. For such unit-generators, we link the audio vector node to the previous node so that it can retrieve the necessary information from the result of the computation in the previous node (Figure 3 middle). Yet, this case also doesn't require the computation of output samples during the generation phase. Just linking these nodes suffices in the generation phase.

We use reference counting [18, p.43] to avoid releasing such an audio-vector node that is still required by another audio-vector node. When generating a new audio vector tree, we give all the node to a reference count of one, neglecting multiple references within the same audio vector tree (one unit generator may be referenced from two or more unit generators) except those nodes that will be linked from the nodes in the next DSP cycle (the linked nodes will be given a reference count of two). This helps avoid a circular reference in the audio-vector tree that causes memory leak, as the reference counting garbage collection cannot release objects in circular references [18, p.24]. Note that no problem is caused in garbage collection by setting reference counts of all the nodes to one in the generation phase. The nodes in the same audio-vector tree are disposed altogether, except the nodes referenced from the next cycle. When we release audio-vector trees, all the nodes in a tree decrease the reference counts just by one. This can be easily performed by visiting each node just once in the depth-first traversal.

Another case to consider is unit generators that involve random values during sound synthesis. As we describe in a later section, for the extension of our implementation technique in the future, it is significantly desirable to produce identical output whenever output samples are evaluated (in other words, we want to guarantee the referential transparency[3] when we evaluate audio-vector trees. This also requires each unit generator's DSP method to evaluate an audio-vector tree in each unit-generator object to be implemented to guarantee the referential transparency). A typical example of this kind is a white noise generator. To achieve such a feature of referential transparency, we retain the seed value for a pseudo random-value generator in the audio vector nodes, with other sound synthesis parameters. The seed value itself is generated by another random-value generator. Given the same seed value, a pseudo random-value generator returns the identical sequence of random values. Thus, the output samples of the unit-generators can be always identical. This case also does not involve any computation of output samples.

3.3 Evaluating Audio-vector Trees

The next phase in our technique is the evaluation of audio vector trees to obtain output samples. This phase is virtually equivalent as the traditional implementation to directly compute output samples of a unit-generator graph. We simply traverse audio-vector trees in depth-first order to compute output samples. While the audio-vector trees are constructed ahead of the time, the computation can utilize the most recent audio input samples, as this phase is lazily evaluated within the audio thread. The result of the computation is memoized within the node to reuse it as it may be required by the linked audio-vector nodes in the next DSP cycle.

3.4 Releasing Audio-vector Trees

After output samples are utilized for sound synthesis, audio-vector trees must be released. As noted earlier, we decrease the reference counts of each node in depth-first order by one, avoiding visiting the same node twice. If the

[3] "A language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program." [19, p.78]

reference count becomes zero, the node is returned to its object pool. If a node still has a reference count larger than one, it is still required from the linked audio-vector node[4].

4. PRELIMINARY EVALUATION

We performed a simple preliminary investigation regarding how much computational overhead may be imposed by our implementation technique. In our experiment, we utilized 256 samples for the audio-vector size, and four audio vector trees are generated ahead of the actual computation. These 1024 (256 * 4) samples correspond to the roundtrip latency about 23 ms. We consider this latency acceptable since it is within the threshold for a just noticeable difference for gestural sound control without tactile feedback [9].

Task #1: 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 #2: FM Synthesis
Ten simple FM synthesis instruments are created. The unit-generator graph of this FM synthesis instrument is shown in Figure 4.

Table 1. Two test tasks for the preliminary evaluation

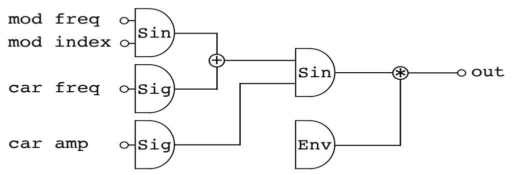

Figure 4. A simple FM synthesis instrument.

Each task generates 10 second output of 64 instruments for the performance measurement. We repeated each task 10 times and obtained the average CPU time costs. Table 2 shows the comparison between our technique and a simple (traditional) implementation that directly generates output samples from a unit-generator graph. We performed two simple sound synthesis algorithms as described in Table 1. The test tasks were performed on a Macbook Air Mid 2011, 13-inch, Intel Core i7 1.8GHz, 4GB Memory, OS X El Capitan. The code was complied with the '*-Ofast*' option (the fastest-aggressive optimization) with the Apple LLVM7.1 compiler.

As shown in Table 2, the performance efficiency in the sound synthesis phase is almost equivalent, as expected. Note that these numbers in the table describe the average time costs for one DSP cycle, not the entire time cost for the sound synthesis. As the worst-case execution time for one DSP cycle can be important for real-time sound synthesis, we also describe max/min time costs for sound synthesis. Both are also almost equivalent. The overhead caused by the tree-generation phase and tree-release phase is around 30% of the sound synthesis phase in this experiment. However, these phases are performed outside the audio thread and do not impose the time cost for sound synthesis; hence, it can be expected that the overall performance efficiency in real-time sound synthesis is equivalent.

5. DISCUSSION

5.1 Overall Benefit

The motivation behind our study is to develop an implementation technique that is appropriate for parallelization, distributed computing, and speculative computation in sound synthesis, as discussed in the later section. However, even with the simple implementation as we described in this paper (still without the extension for parallelization/distribution/speculation in sound synthesis), it is still a benefit to separate the audio thread and control task thread, maintaining synchronous behavior and the minimum roundtrip latency, while making real-time sound synthesis more stable by amortizing the time costs for control tasks. Thus, our technique provides a practical solution for the tradeoff between the roundtrip latency and stability in real-time sound synthesis in a synchronous computer music system.

The issue regarding the action-to-sound latency in interaction remains, similarly as the ring buffer implementation. Yet, in our technique, as the threads for control tasks do not have to compute output samples, there is some potential to significantly reduce the action-to-sound latency in interaction in comparison with the ring buffer implementation. Even when the FIFO queue of audio-vector trees becomes almost empty, the audio-vector trees can be provided to the queue before the audio thread starves much faster than the computation of output samples in the ring buffer implementation. The generation of audio-vector trees can be performed much faster than the computation of output samples (as discussed later, the generation and release phases take only less than one-third of the CPU time costs for the computation of output samples).

5.2 Computational Overhead

Our technique requires two extra phases (the generation phase and the release phase). However, these phases do not impose a large computational time cost in comparison with the computation of output samples and both phases are not performed in the audio thread where sound synthesis is performed, but in the control task thread. Control tasks are normally scheduled quite sporadically, and it is very rare to schedule any task in every DSP cycle. In the case in which the CPU time required for sound synthesis is 100% for the given deadline, the 30% of this time cost will be imposed to the control thread, and the rest 70% of will be

Algorithm	**sound synthesis phase**			**tree generation phase**			***tree release***		
	avg.	***min***	***max***	***avg.***	***min***	***max***	***avg.***	***min***	***max***
Task 1: Additive Synthesis									
traditional	0.715	0.660	2.033						
audio-vector trees	0.708	0.661	1.948	0.109	0.089	0.375	0.097	0.075	0.319
Task 2: FM Synthesis									
traditional	0.865	0.800	2.248						
audio-vector trees	0.894	0.809	2.362	0.139	0.123	0.420	0.121	0.086	0.322

Table 2. The test results for the performance measurement (all the numbers are in milliseconds).

the available time cost for control tasks. If the same amount of the CPU time must be used for control tasks in the simple audio thread implementation or the ring buffer implementation, it requires 170% of the available time for one DSP cycle, as control tasks and sound synthesis must be performed in the same thread. This clearly causes a problem in real-time sound synthesis. However, our implementation technique may still meet the real-time deadline safely, if they are executed parallelly on different CPU cores.

5.3 Memory Overhead

As our technique generates many audio-vector nodes, for which sound synthesis parameters and an audio vector are coupled, the overall memory consumption in our technique can be larger than the traditional implementations. However, the memory space consumed by sound synthesis parameters is not very large for modern computer systems, as the number of the sound synthesis parameters is normally not large. While audio vectors may consume some more memory (for instance, if we use the *float* type in C and the audio vector size is 256, it consumes 4 * 256 bytes = 1Kb for every audio-vector node), these are still not very large in comparison to image processing applications.

In the case that memory consumption is of significant interest, for example, for sound synthesis on an embedded system with scare memory resources, the on-stack computation technique that we developed can be utilized. This technique utilizes the characteristics in unit-generator-based sound synthesis in which most unit-generators do not reuse output samples after the last use in the same DSP cycle. Hence, each unit-generator does not have to retain an audio vector and utilize audio vectors allocated temporarily on stack when sound synthesis is performed. This technique can be directly applied to our technique so that the allocation of audio vectors can be avoided in most nodes. Thus, memory consumption in our technique can be significantly reduced when combined with the on-stack allocation technique. Only the nodes linked from the next DSP cycle are required to allocate audio-vectors in the heap area.

6. EXTENDED DISCUSSION

6.1 Update-caching Technique

The update-caching technique that we previously developed also allows a separation between the audio thread and control task thread, while performing sound synthesis in the audio thread but still maintaining synchronous behavior. In comparison with update caching, our new technique imposes extra time costs for the tree-generation phase and the tree-release phase. However, these time costs would not be crucial for control task thread, as control tasks are normally scheduled quite sporadically (and not scheduled very frequently). Another merit of update caching is that it realizes the sample-rate accurate control of sound synthesis parameters without significant performance damage. Yet, it should be noted that our technique can also be combined with the update-caching technique, simply by linking cached update events for each audio-vector node. Moreover, update caching cannot be simply parallelized as discussed in the next section, since update-caching still must sequentially compute output samples for the current DSP cycle before advancing to the next; hence, update caching may require more consideration for parallelization.

6.2 Parallelization

As the audio-vector tree in our technique is a tree data structure that is lazily evaluated, trees can be generated without performing the computation of output samples. It should be also noted again that sound synthesis parameters in each node are sufficient to perform the computation without external parameters, and the output is always identical whenever it is computed. Hence, it is possible to evaluate each audio vector tree independently from other trees. Even when a certain audio vector node requires the result of the computation in the previous cycle, it can traverse the link to the previous node and then lazily evaluate the previous node. Such a feature is potentially beneficial to parallelize sound synthesis as each audio-vector tree can be evaluated by multi-threading without a data race. There is no need to lock each audio-vector node during the computation and, even when two or more threads happen to compute the same audio vector tree simultaneously, the output samples are guaranteed identical; hence, our implementation technique provides ideal features for parallelization.

When the control task thread has no task scheduled in the cycle, the thread can utilize its CPU time to actively evaluate the audio-vector trees before the audio thread evaluates them (note that audio-vector trees can even be just partially evaluated except those nodes that depend on the audio input). Even when multiple threads happen to evaluate the same audio-vector tree simultaneously, no data race is caused, and the output will be identical. It is also possible to allocate other high-priority threads, which are assigned to different CPU cores, to actively evaluate audio-vector trees in the audio-vector tree queue.

Such an extension of our technique for parallelization can still maintain precise synchronization between control tasks and sound synthesis in logical time, as all control tasks are performed within the same control thread.

6.3 Distributed Sound Synthesis

As the audio-vector trees are lazily evaluated and return identical output samples whenever (and wherever) evaluated, our technique is potentially beneficial for distributed sound synthesis. Audio-vector trees can be distributed to other computers in the network with the timestamp in logical synchronous time. While this may increase the audio latency as it involves the network communication, such an extension can be beneficial to distribute CPU-intensive real-time sound synthesis tasks among networked computers, while keeping the high timing precision in compositional algorithms. If the output samples of the entire systems must be synchronized, the time-stamped output from each computer can be gathered into one node for the audio output.

6.4 Speculative Sound Synthesis

In [1], we described the speculative sound synthesis technique, in which sound synthesis is speculatively performed with the optimistic assumption that there would be no change given to sound synthesis parameters. When any update is performed (when the speculation failed), the output samples are recomputed from the timing when the speculation failed. We also discussed that this technique can be especially beneficial when the speculation is performed over several DSP cycles for the future output by multi-threading, while we still have not extended this technique for such inter-DSP-cycle speculation.

Our new implementation technique would be potentially desirable for such speculation over several DSP cycles, as it can minimize the penalty for the failure in speculation. As the audio-vector trees are lazily evaluated, even if we speculate for several DSP cycles, the penalty time cost for speculation failure is mainly for those audio-vector trees already speculatively evaluated and not large for those trees that are still unevaluated. When the speculation failed, it only suffices to replace the old speculated audio-vector trees with new ones with updated sound synthesis parameters. This replacement would not impose much time cost.

7. CONCLUSIONS AND FUTURE WORK

We developed an implementation technique that views a unit-generator graph as a generator of lazily evaluated audio-vector trees. The simplest implementation of this technique allows us to perform sound synthesis and control tasks into different threads while maintaining a synchronous behavior. Even in this simplest implementation, our technique helps to solve the tradeoff in a synchronous computer music system between the roundtrip latency (in the audio input/output) and the stability in real-time sound synthesis (by amortizing the time cost for control tasks over DSP cycles), while the action-to-sound latency remains. Some overhead may be imposed for the generation and release phases of audio-vector trees to the control task thread, yet the total time cost for these phases would not be crucial, especially because control tasks are normally sporadically scheduled, and it is quite rare to schedule any CPU intensive task in every DSP cycle.

We also extended the discussion that our technique is potentially quite beneficial to parallelize, distribute, or speculate sound synthesis. We plan this extension as a future work. Such an investigation into implementation techniques for high-performance sound synthesis techniques is of significance today, as Moore's law may end soon.

8. REFERENCES

[1] H. Nishino et al., "Speculative digital sound synthesis," Proc. of Sound and Music Computing, 2016.

[2] M. Waldrop, "The chips are down for Moore's law," Nature News, 2016

[3] C. Adamson et al., Learning core audio: a hands-on guide to audio programming for Mac and iOS, Addison-Wesley Professional, 2012.

[4] D Philips, Linux music & sound, No Starch Press, 2000.

[5] Microsoft, About WASAPI (Windows), https://msdn.microsoft.com/enus/library/windows/desktop/dd371455.aspx (accessed on Mar/28/2018).

[6] Steinberg, ASIO 2.0 Audio Streaming Input and Output Development Kit., Steinberg Media Technologies, 2006.

[7] A. Burns and A.J. Wellings, Real-time systems and programming languages: ADA 95, Real-time Java, and Real-time Posix, Addison Wesley, 2001.

[8] S., A. Finney, "Auditory feedback and musical keyboard performance," Music Perception, 15 (2), 1997.

[9] T. Mäki-Patola et al., "Latency tolerance for gesture controlled continuous sound instrument without tactle feedback," in Proc. ICMC, 2004.

[10] H. Nishino., "Update-caching technique for unit-generator-based sound synthesis,", in Proc. ICMC 2017

[11] R. B. Dannenberg et al., "Fugue: a functional language for sound synthesis," in Computer, 24 (7), 1997.

[12] H. Nishino et al., "Lazy evaluation in microsound synthesis," in Proc. Sound and Music Computing, 2016.

[13] R. B. Dannenberg et al., "The implementation of Nyquist, a sound synthesis language," in Computer Music Journal, 21(3), 1997, pp.71-82.

[14] E. Brandt, "Temporal type constructors for computer music programming," PhD dissertation, Carnegie Melon University, 2008.

[15] C. Roads. Microsound. MIT Press. 2004.

[16] H. Nishino, LC: Mostly-strongly-timed computer music programming language that integrates objects and manipulations for microsound synthesis, PhD dissertation, National University of Singapore, 2014.

[17] A. Freeman., "The object pool pattern." In *Pro* Design Patterns in Swift. Apress, Berkley, 2015.

[18] R. Jones et al. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, 1996.

[19] J.C. Mitchell. Concepts in Programming Languages, Cambridge University Press, 2011.