Update-caching Technique for Unit-generator-based Sound Synthesis

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

ABSTRACT

This paper describes 'update caching', a novel implementation technique for unit-generator-based sound synthesis that we developed. The trade-off between the computational efficiency benefit by the utilization of audio vectors and the resulting damage to timing precision in updating sound synthesis parameters is one of the most well-known problems in computer music software design. Our new technique first processes all the tasks in a DSP cycle and caches update events of sound synthesis parameters with timestamps. Then, the cached events are processed inside the signal processing loops of unit-generators during the computation of output samples. In most practical situations, such a technique allows sample-rate accurate updates of sound synthesis parameters without significant damage to the computational efficiency. While it does not improve the minimum feedback time, which is another problem caused by audio vectors, our technique provides a practical solution to the long-standing trade-off between computational efficiency and timing accuracy in unit-generator-based sound synthesis. Such an investigation into a novel implementation technique would be beneficial for the research and development of next-generation computer music systems.

1. INTRODUCTION

The computational efficiency in sound synthesis is still a significant issue in the development of computer music software today, as Moore's law may end soon [1], and it would be too optimistic to assume the continued rapid improvement of computational speeds of CPUs as in the past decades. Thus, there is a strong necessity to investigate a new implementation technique that improves the computational efficiency in sound synthesis.

One of the most widely used implementation techniques for unit-generator-based sound synthesis is to utilise audio vectors [2, p.467]. The utilisation of audio vectors can lead to significant improvement in performance efficiency. However, the involvement of audio vectors normally leads to the existence of a control rate, which hinders the sample-rate accurate update of sound synthesis parameters through a user program. Yet, as human auditory perception is not very sensitive to some aspects of sounds, such as changes in loudness or pitch contours, the lack of sample-rate accuracy in updating sound synthesis parameters has been generally considered acceptable in past decades.

However, the emergence of novel sound synthesis techniques and creative practices in computer music raises the question regarding whether the lack of sample-rate accuracy is still acceptable today. Recently, computer music languages are often developed with sample-rate accurate timing behaviour as one of the important criteria in the design [3] [4] [5].

Hence, we developed *update caching*, a novel implementation technique that allows the sample-rate accurate control of sound synthesis parameters, while still receiving the performance efficiency benefit of the utilisation of audio vectors. This implementation technique divides sound synthesis into two stages. In the first stage, all the tasks scheduled in the DSP cycle are processed. No sound synthesis by unit-generators are performed at this point; yet, all the updates made to sound synthesis parameters by these tasks are cached within unit-generators with timestamps (in logical time) for when the updates should be performed. In the second stage, the output samples for the same DSP cycle are computed using audio vectors. When unit-generators compute their outputs, the cached update events are applied at the timing that their timestamps specify. As these updates are performed within the processing loops to compute the audio vectors of the output samples, the update of sound synthesis parameters can be performed with sample-rate timing accuracy. This technique is applicable for both non-real-time and real-time sound synthesis.

As we describe in a later section, the computational overhead imposed by our implementation technique is quite trivial in most practical situations. Thus, the sample-rate accurate control in sound synthesis can be achieved without significant damage in comparison with the traditional audio-vector implementation. However, since the samples are routed between unit-generators in the unit of audio vectors, this technique does not contribute to improving the minimum feedback time.

As mentioned, the computational speed of CPUs cannot be expected to continue improving as it has in past decades; therefore, such an implementation technique would be

quite beneficial both for support of creative musical practices of our time and for further research and development of next-generation computer music languages and systems.

2. RELATED WORK

2.1 Audio Vectors

```
01:float osc(float *output, float amp, float freq,
02:          float *table, float *index,
03:          int len, int vecsize, long sr)
04:{
05:  // increment
06:  float incr = freq * length / sr;
07:
08:  // processing loop.
09:  for (int i = 0; i < vecsize; i++){
10:      // truncated lookup
11:      output[i] = amp * table[(int)(*index)];
12:      *index += incr;
13:      while(index >= len) index -= len;
14:      while(index < 0) index += len;
15:  }
16:  return *output;
17:}
```

Figure 1. Lookup table oscillator function that utilises audio vectors [2, p.466].

Generally, every time a function call is made, a computer program must prepare a new *stack frame* (or *activation record*) for the return address, local variables, parameters, and other temporaries[1] [6, chapter 7]. A function call also involves other behaviours, such as the jump to the address of the function and the memory access to load the instructions from the main memory to the CPU cache. Hence, if one function call is made to compute each output sample, it can impose the time cost on the runtime performance in sound synthesis.

Yet, the audio-vector implementation (as shown in Figure 1) can reduce such overhead and leads to improvement in the computational efficiency[2] since input and output samples are given/computed in the unit of audio vectors (a vector of samples), and the audio vectors are processed at once in the internal processing loop. Just one function call is necessary to compute these output samples, the size of which is the length of the audio vectors. Indeed, this technique has been utilised since the early stage of computer music history. Music V, which is known to be the first language that utilised 'block processing of data' (or audio vectors) was developed in 1966 [7].

Vercoe further extended this technique, finding that 'up to 50% of musical signal processing is aimed at shaping loudness and pitch contours, functions essentially of acoustic control that need not be controlled at audio rates' [8], by observing the use of the MUSIC 360 language [9]. He developed Music 11 [10] in 1987, the first computer music language that introduced the distinction between the audio signal and control signal. By updating the control signal at a lower rate, further improvement of the computational efficiency was achieved.[3]

2.2 Audio Rate and Control Rate

The utilisation of audio vectors can lead to the distinction between the audio rate and the control rate. For instance, in the Figure 1 example, the parameter *amp* or *freq* can be given only when the function is called and cannot be changed while the audio-vector output is being computed inside the processing loop (in other words, *amp* and *freq* can be updated only at the control rate).

The existence of control rates has been generally considered acceptable in the past decades. As described in the previous section, some parameters, such as loudness and pitch contours, do not have to be computed at the audio rate, since human ears are not sensitive to such parameters.

Yet, the sample-rate accuracy (or the audio-rate accuracy) in task scheduling and sound synthesis control has become an important criterion for computer music systems today, as it is sometimes crucial for creative musical practices in our time. For example, many microsound synthesis techniques [11] are algorithmic. The lack of sample-rate accuracy in scheduling tasks and microsounds can lead to output samples that are different from what is theoretically expected, and the differences are often perceptible to human ears. For another example, as Lyon discussed in [12], '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 change in alignment on the order of a couple of milliseconds will create different comb filtering effects, and the transients slightly realign on successive attack'.

Indeed, such demands for the sample-rate accurate timing behaviour are already reflected in the design of recent computer music languages. For instance, such languages as ChucK [3], LuaAV [4], and LC [5] provide the sample-rate accurate timing precision in both task scheduling and sound synthesis parameter control. Yet, to achieve this sample-rate accurate timing precision, these languages compute sample-by-sample and abandon the use of audio vectors in sound synthesis. This results in significant damage to the real-time sound synthesis performance efficiency. As stated, it is unlikely for the computational speed of commercially available CPUs to continue improving as it has in the past. Thus, it is necessary to investigate how software can help further improve the performance effi-

[1] While optimization techniques, such as *inline function expansions*, can reduce the overhead of the stack frame allocation, many techniques are often not applicable to computer music languages, as unit-generator synthesis graphs are normally created dynamically at runtime.

[2] While the performance efficiency largely depends on the actual implementation of a computer music system, our previous work [13] describes that audio-vector implementation can be about three times faster than the sample-by-sample computation in the simple test systems (with audio vector size of 256 samples).

[3] Music 11 used scalar values for control signals, while its predecessor languages (such as Music V) still used audio vectors for all signals [13].

ciency of computer music systems, while providing sample-rate accurate timing precision in the control of sound synthesis.

2.3 Speculative Sound Synthesis

It seems that the trade-off between computational efficiency and timing precision in sound synthesis is taken for granted even today, and how we can solve this trade-off is still a largely unexplored issue in computer music software design. The rapid improvement of the CPU speeds in the past several decades may have overshadowed this problem, or researchers may have had more interest in improving performance efficiency using general purpose GPUs (GPGPUs) or just-in-time compilation as in [14] [15]. However, it is still true that sample-by-sample computation significantly damages the runtime performance in many computer music languages and systems.

Our past research work, *speculative sound synthesis,* also investigates the solution of this issue by an implementation technique. In [13], we described an implementation technique that speculatively computes output samples. In the speculative sound synthesis technique, the output audio samples for one DSP cycle are computed by the audio vector at once at the beginning of the cycle, assuming there will be no update to the sound synthesis parameters. Then, the scheduled tasks are performed with sample-rate accurate timing precision (in logical time). When any update to the sound synthesis parameters occurs, the future output samples of the sound objects (the unit-generator graph) will be recomputed, while the past samples (in the same audio vectors) from the update time remain unchanged.

Our prototype implementation shows that there seems to be no significant damage to the performance efficiency by speculation. However, it should be noted that this technique largely depends on the assumption that the updates to the sound synthesis parameters are rare in most practical solutions (in that updates are not performed many times in one DSP cycle). Moreover, the cost of recomputation of speculative sound synthesis can be significant if there are too many updates performed in one DSP cycle. This technique also does not contribute to relaxing the limitation of the minimum feedback time caused by the audio vectors, as the samples are routed between unit-generators in the unit of the audio vector.

3. UPDATE-CACHING TECHNIQUE

3.1 The Algorithm

Our update-caching technique has a motivation to provide computational efficiency and sample-rate accurate timing behaviour simultaneously, similar to the speculative sound synthesis technique. However, the update-caching technique significantly differs from speculative sound synthesis in that it does not recompute the output samples at all. Hence, a large potential computational overhead when too many updates are made in one DSP in speculative sound synthesis can be avoided.

Our implementation technique divides sound synthesis into two stages. In the first stage, it performs all the tasks scheduled in the current DSP cycle. Yet, all the updates performed to unit-generators are cached within the unit-generators with the timestamp in logical time (measured in samples). In the second stage, unit-generators perform sound synthesis, utilising audio vectors. However, the cached update events are also processed 'within the processing loop', when the given timestamps are reached. Thus, while the utilisation of audio vectors can contribute to improving the computational efficiency, scheduled tasks and updates to sound synthesis parameters can be also performed with sample-rate accuracy. The flowcharts in Figures 2 and 3 show how the DSP cycle function processes and how an update-caching unit-generator computes output samples.

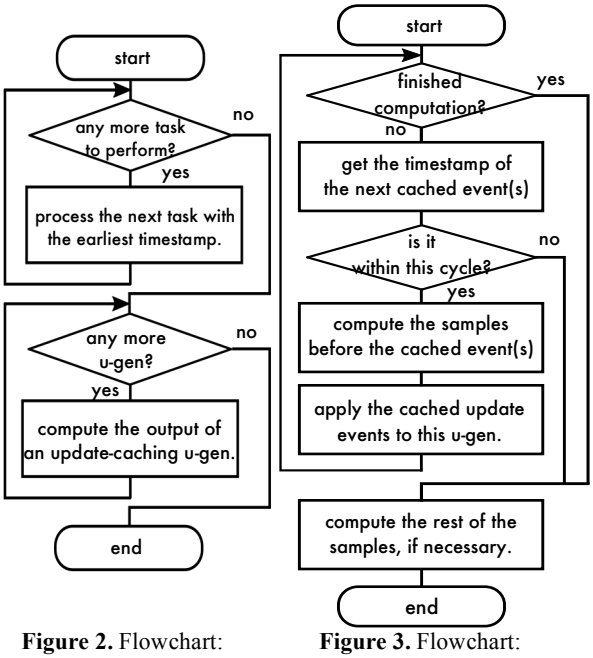

Figure 2. Flowchart: DSP cycle function.

Figure 3. Flowchart: update-caching unit-generator.

3.2 The Implementation

Figures 4 and 5 show the code excerpts of an update-caching sine-wave oscillator unit-generator taken from our prototype implementation for the evaluation.

```
01:class UCSineOsc:public UpdateCachingUGen
02:{
03:public:
04:  UCSinOsc(int64_t avecSzie,
05:           double freq, double amp);
06:  virtual ~UCSineOsc(void);
07:
08:  virtual double* compute(int64_t now);
09:  virtual double* getOutput(void);
10:  virtual void    update(UpdateMessage* msg);
11:
12:  void setFreq(AudioEngine* engine,
13:               int64_t now, double freq);
14:  void setAmp (AudioEngine* engine,
15:               int64_t now, double amp);
16:}
```

Figure 4. Code excerpt of an update-caching sine-wave oscillator unit-generator (the class definition).

```
01:double* UCSineOSc:compute(int64_t now)
02:{
03:  phase = fmod(phase, 2 * PI);
04:
05:  int64_t end = now + this->avecSize;
06:  int64_t i = 0;
07:  while (now < end){
08:    //any event to process in this cycle?
09:    int64_t evtime = this->getNextEventTime();
10:    if (evtime < 0 || evtime >= end){
11:      //if there's no event, exit this loop.
12:      break;
13:    }
14:    //compute samples until the next event(s).
15:    for (  ; now < evtime; now++){
16:      this->output[i++] = sin(phase) * amp;
17:      phase += phaseInc;
18:    }
19:    //apply cached update event(s) at this timing.
20:    this->applyCachedUpdateEvents(evtime);
21:  }
22:
23:  //compute samples left uncomputed if any.
24:  for (  ; now < end; now++){
25:    this->output[i++] = sin(phase) * amp;
26:    phase += phaseInc;
27:  }
28:  //computed all the samples for this DSP cycle.
29:  return this->output;
30:}
31:
32:void UCSineOsc::update(UpdateMessage *msg)
33:{
34:  //this function is called when any cached
35:  //update event must be processed.
36:  switch(message->pnum){
37:    case FREQ:
38:      this->freq = msg->v.d;
39:      phaseInc = 2 * PI * freq / GetSampleRate();
40:      break;
41:    case AMP:
42:      this->amp = msg->v.d;
43:      break;
44:    default:
45:      break;
46:  }
47: return;
48:}
49:
50:void UCSineOsc::setFreq(AudioEngine* engine,
51:                         int64_t now, double freq)
52:{
53:  //get a message object from the object pool.
54:  UpdateMessage* m = this->pool->getObject();
55:
56:  //set up the message and cache it.
57:  m->time = now;
58:  m->pnum = FREQ;
59:  m->v.d  = freq; //'v' is a union of double/int.
60:  this->cacheUpdateMessage(m);
61:  return;
62:}
```

Figure 5. Code excerpt of an update-caching sine-wave oscillator unit-generator (the class implementation).

As shown in these figures, the implementation can be quite simple and most implementations can be inherited from the superclass (UpdateCachingUGen) and shared by all the unit-generator classes. When an update is performed (as shown in lines 50-62 in Figure 5), the tasks for the sound synthesis control will call a method to update the parameter with the current logical of the task (*now*). Yet, the unit-generator just caches the update events internally. When the unit-generator performs sound synthesis, it first checks if there is any cached event in the future within the DSP cycle and computes the output samples until the timestamp of the update events, and then applies the cached updates (line 20). When the underlying implementation in the superclass applies the update, it calls the *update* method of the unit-generator class (lines 32-48), where the actual update of sound synthesis parameters is performed.

The main processing loop continues this procedure until it reaches the end of the DSP cycle or no more cached update events are found in the same DSP cycle. When there are no more cached update events within the same DSP cycle, it computes the rest of the output samples (lines 24-27).[4] Thus, since the update events of sound synthesis parameters are performed within the processing loop, they can be performed with sample-rate accuracy in logical time. The instantiation of a new sound object with sample-rate accurate timing can be realised by just providing zeros for the output samples before the timestamp of the instantiation.

4. EVALUATION

4.1 Testing Environment

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. All of them share the same frequency, which is randomly updated every 2, 4, 8, 16, 32, and 64 samples.
Task #2: FM Synthesis
Ten simple FM synthesis instruments are created. The unit-generator graph of the FM synthesis instrument is shown in Figure 6. All of them share the same frequencies (modulator and carrier), which are randomly updated every 2, 4, 8, 16, 32, and 64 samples.

Table 1. Test tasks for the performance measurement.

Table 1 shows our test tasks. We compared three implementations to measure computational efficiency: sample-by-sample implementation, audio-vector implementation, and update-caching implementation. Each task is performed to update the sound synthesis parameters 1, 2, 4, 8, 16, and 32 times in every 64 samples, which is the audio vector size.

The test system was implemented in C++ and evaluated on Mac OS X.[5] We used the *clock* library function to measure the CPU time spent only for the sound synthesis part of the code, excluding the effect of task switching by the operating system.[6] As the test tasks involve random values, we used the C++ *std::mt19937*, a *Mersenne Twister* random-value generator class. To guarantee that the exact sequence of random values is shared among all test tasks, the same seed value is used. Each task was performed five times, and the average/min/max CPU times were obtained. The

[4] This part of the code can be simpler if we move the checks and actual updates of cached events into the superclass and leave only the processing loop in a subclass. The code is presented in this implementation for terser presentation to readers since it is easier to grasp what is performed in our update-caching technique in this way.

[5] Mac OS X 10.11.6 (El Capitan) on the MacBook Air 11 inch (early 2015, Core i5 1.6 GHz, 4 GB RAM). The Apple LLVM compiler is used with the compile option '-Ofast'.

[6] However, the time cost of cache misses is still included. Cache misses caused by other processes can be hardly controlled by a user program.

maximum CPU time is measured to imply the worst-case execution time expected for real-time sound synthesis.

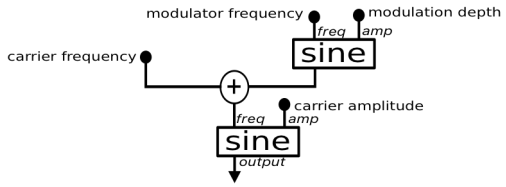

Figure 6. A simple FM synthesis instrument.

4.2 Test Results

Algorithm	*Avg. (msec)*	*Max (msec)*	*Min (msec)*
Task 1: Additive Synthesis			
Update Interval: 64 samples			
sample-by-sample	0.152	0.649	0.088
update caching	0.078	0.437	0.044
audio vector	0.076	0.342	0.043
Update Interval: 32 samples			
sample-by-sample	0.144	0.723	0.090
update caching	0.080	0.397	0.046
Update Interval:16 samples			
sample-by-sample	0.147	0.730	0.089
update caching	0.089	0.460	0.051
Update Interval: 8 samples			
sample-by-sample	0.157	0.622	0.092
update caching	0.099	0.432	0.058
Update Interval: 4 samples			
sample-by-sample	0.158	0.892	0.096
update caching	0.129	0.416	0.074
Update Interval: 2 samples			
sample-by-sample	0.167	0.658	0.099
update caching	0.172	0.559	0.110
Task 2: FM Synthesis			
Update Interval: 64 samples			
sample-by-sample	0.083	0.467	0.047
update caching	0.040	0.194	0.020
audio vector	0.042	0.217	0.021
Update Interval: 32 samples			
sample-by-sample	0.081	0.371	0.048
update caching	0.042	0.238	0.023
Update Interval:16 samples			
sample-by-sample	0.085	0.424	0.048
update caching	0.047	0.257	0.025
Update Interval: 8 samples			
sample-by-sample	0.086	0.347	0.049
update caching	0.053	0.440	0.027
Update Interval: 4 samples			
sample-by-sample	0.086	0.441	0.051
update caching	0.067	0.392	0.037
Update Interval: 2 sample			
sample-by-sample	0.106	0.537	0.058
update caching	0.091	0.293	0.053

Table 2. Performance measurement result.

Table 2 shows the measurement results (average/max/min CPU time costs of the entire DSP cycle). The graphs in Figures 7 and 8 illustrate the average CPU time per DSP cycle for the additive synthesis task and FM synthesis task, respectively. As the audio-vector implementation cannot update sound synthesis parameters at sample-rate accuracy, its CPU time cost is shown only for the update interval of 64 samples, which is the audio vector size utilised in these test tasks. The '*update interval*' describes how often the sound synthesis parameters were updated in the test tasks. For instance, with the update interval of 16 samples, the updates were performed four times in every DSP cycle since the test tasks utilised the audio vector of 64 samples for the DSP cycle.

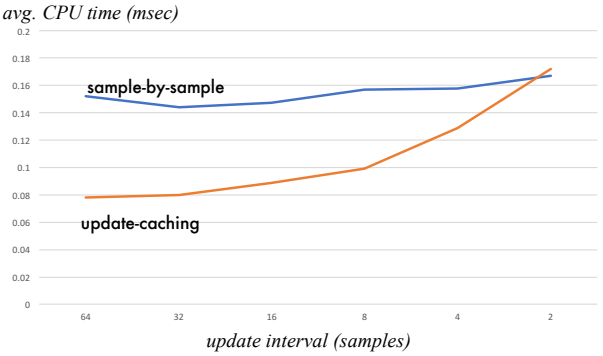

Figure 7. Average time cost for the additive synthesis task.

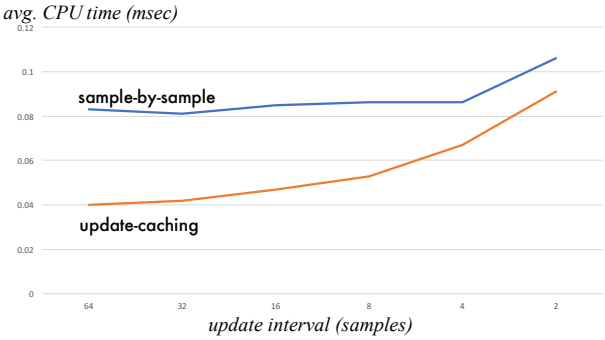

Figure 8. Average time cost for the FM synthesis task.

5. DISCUSSION

As clearly shown in Table 2, in our test environment, the audio-vector implementation is about twice as fast as the sample-by-sample implementation for both test tasks (see *update interval: 64 samples* in the table). The time cost for the sample-by-sample implementation stays almost the same, regardless of the update interval. This is a reasonable result, as it always behaves the same (compute one sample at a time), whatever the update interval is.

On the contrary, for the update-caching implementation, the performance for the update interval of 64 samples is almost as fast as the audio-vector implementation. This is something fairly expected; the update-caching implementation will behave almost the same as the audio-vector implementation when there is only one update event in a DSP cycle. The update-caching implementation maintains the good performance efficiency even when more updates are performed, which is comparable to the audio-vector implementation, while it still maintains sample-rate accurate timing behaviour. For instance, even at the update interval of eight samples, the CPU time utilised for sound synthesis is still slightly more than half of the sample-by-sample implementation. The same tendency is observed for the maximum CPU time, and this also implies a significant improvement in the worst-case execution time. This characteristic is favourable to avoid missing the deadline for real-time sound synthesis.

However, as clearly exhibited in Figures 7 and 8, as the number of updates performed in one DSP cycle increases, the update-caching implementation seems to lose such

benefits of the audio-vector computation. Yet, unlike *speculative sound synthesis*, as the recomputation of the output samples is not performed in update-caching even in a situation with very frequent updates, significant damage to the performance efficiency can be avoided. However, too much recomputation may be performed in speculative sound synthesis in the same situation.

Moreover, it should be noted that such highly frequent updates of sound synthesis parameters (e.g., every two or four samples) are rare in most practical situations. Even though it may occur for some reason, it may be not too unfair to assume that such frequent updates are rarely applied to many unit-generators at once. Hence, the reduction of the CPU time cost by update caching for other unit-generators, for which the synthesis parameters are not frequently updated, may compensate for the overhead. In contrast, in the sample-by-sample implementation, all unit-generators impose the overhead to compute one sample at a time; thus, the entire time cost could be still significantly reduced by our update-caching technique in comparison with the sample-by-sample implementation.

6. FUTURE WORK

Since the update-caching technique caches update events with a timestamp (in logical time) and performs scheduled tasks in a different stage from the sound synthesis stage, scheduled tasks can be performed in a separate thread from the audio thread/audio callback function. This suggests that, when the audio thread or callback function is activated, it does not have to perform scheduled tasks (as they can be performed in a separate thread before output samples are required for DAC output). Then, the CPU time before the deadline for real-time sound synthesis can be fully utilised for sound synthesis, which is favourable for the CPU intensive sound synthesis technique.

Furthermore, the latency between audio input and output can be minimised, if the audio callback function also receives the input samples together when the output samples are requested (as seen in Apple's CoreAudio framework) since the input samples of the same DSP cycle can be directly used for the output without buffering the input into a buffer. Such a separation between the audio thread and task threads with sample-rate accurate timing behaviour would be also beneficial when external DSP hardware or a GPGPU is utilised for a computer music system. Even without the precise synchronisation between the CPU and external DSP hardware (or a GPGPU), control tasks and sound synthesis can be coordinated with the sample-rate accuracy in logical time; we leave this investigation for our future work.

7. CONCLUSIONS

In this paper, we described *update caching*, a novel implementation technique for sound synthesis. The technique performs control tasks in the first stage, yet just 'caches' update events without applying them immediately to unit-generators. Then, cached events that are processed during the output samples are computed in the second stage. As the control events are processed inside the processing loop of the output audio vectors, the sample-rate accuracy in timing behaviour in logical time can be achieved, while maintaining the computational efficiency of the audio-vector implementation without significant damage.

While this technique does not contribute to improving the minimum feedback time, which is another problem caused by the utilisation of audio vectors, it can achieve computational efficiency and sample-rate accurate timing behaviour at the same time. Such a characteristic is considered beneficial for both the support of contemporary computer music practices and further research for next-generation computer music systems. Our update-caching technique provides a practical solution for the long-standing trade-off in computer music software design.

8. REFERENCES

[1] M. Waldrop, 'The Chips are down for Moore's law', *Nature News,* 2016.

[2] R. Boulanger et al., *The Audio Programming Book*, MIT Press, 2010.

[3] G. Wang, *The chuck audio programming language. A strongly-timed and on-the-fly environ/mentality,* PhD thesis, Princeton University, 2008.

[4] G. Wakefield *et al.,* 'LuaAV: Extensibility and heterogeneity for audiovisual computing', *Proc. of the Linux Audio Conference,* 2010.

[5] H. Nishino, *LC: A Mostly-strongly-timed Prototype-based Programming Language that Integrates Objects and Manipulations or Microsound Synthesis*, PhD thesis, National University of Singapore, 2014.

[6] W. Apple and P. Jens, *Modern Compiler Implementation in Java*. MIT Press, 2002.

[7] B. Vercoe, 'New Dimensions in Computer Music', *Trends & Perspective in Signal Processing*, *2*(2), 1982.

[8] B. Vercoe, 'Computer systems and languages for audio research', *Proc. of Audio Engineering Society Conference*, Audio Engineering Society, 1982.

[9] B. Vercoe, 'The MUSIC 360 language for digital sound synthesis.' *Proc. of the American Society of University Composers*, 6, 1971.

[10] B. Vercoe, Reference Manual for the Music 11 Sound Synthesis Language, *Program Documentation,* MIT Experimental Music Studio, 1981.

[11] C. Roads, *Microsound*, MIT Press, 2004.

[12] E. Lyon, 'A Sample Accurate Triggering System for Pd and MaxMSP', *Proc. of ICMC,* 2006.

[13] H. Nishino et al., 'Speculative Sound Synthesis', *Proc. of Sound and Music Computing*, 2016.

[14] B. Cowan and B. Kapralos, 'Spatial sound for video games virtual environments utilizing real-time GPU-based convolution', *Proc. of Future Play*, *ACM,* 2008.

[15] W. Smith and G. Wakefield, 'Augmenting computer music with just-in-time compilation', *Proc of International Computer Music Conference*, 2009.