ON-STACK COMPUTATION OF AUDIO VECTORS FOR UNIT-GENERATOR-BASED SOUND SYNTHESIS

Hiroki Nishino

Chang Gung University, Dept. of Industrial Design, Taiwan

ABSTRACT

This paper describes a novel implementation technique for unit-generator-based sound synthesis, which allocates audio vectors on the stack frame. The computation is also performed on the stack and the same audio vectors can be shared and reused by sound synthesis modules (e.g., patches or instruments). As the number of audio vectors involved in sound synthesis can be significantly reduced in comparison with a straightforward implementation technique that allocates audio vectors for each unit generator in the heap area, our on-stack allocation technique is desirable for sound synthesis in a memory-limited embedded system. As the utilization of embedded systems is a topic of interest among the NIME (new interfaces for music expression) community, such a memory-efficient implementation technique for sound synthesis can benefit the research and practices in the field.

1. INTRODUCTION

The research and practices in NIME (new interfaces for musical expressions) often involve the utilization of embedded systems. Some NIME instruments consist of an embedded device for external sensors and electric parts (such as LEDs and actuators) and a personal computer for real-time sound synthesis. With this approach to combining an embedded device and a personal computer, NIME instruments can utilize rich computational resources with which a personal computer is equipped (e.g., a faster CPU, large memory resources, HDD/SDD, and an LCD display), yet NIME instruments must accept the latency and in both sound I/O and interaction. There are various factors that increase such latency, the list of which includes audio devices, operating systems, driver software, communication between the personal computer and the embedded device (such as the serial I/O), etc.

On the other hand, there are also NIME instruments that investigate the utilization of embedded devices for sound synthesis (e.g., Satellite CCRMA [1]). The recent popularization of affordable but powerful embedded devices has made it realizable to perform real-time sound processing and synthesis even solely on an embedded device. One of the significant benefits of such embedded systems is that they can reduce the latency in interacting with sensor input and other electric parts, since sound synthesis programs can be directly hosted in the same environment in which sensors and electric parts are controlled.

The computational resources and available user interface devices can significantly vary among such embedded systems. Some embedded devices are more like single-board computers. They can run an operating system and are even equipped with the display output (such as HDMI or DVI-D). For instance, Raspberry Pi [2] belongs to such a category. Yet, some devices are designed to allow user programs to occupy the entire computational resources as possible, just with the layers of the real-time operating system and/or the dedicated software libraries, allowing bare-metal programming. The ARM's mbed family [3] are a typical recent example of this kind. In such an environment, user programs can dedicate most available computational resources to the NIME instrument and can expect a further reduction in the latency. However, the embedded devices of this kind often do not provide large memory resources. In such a memory-limited system, it is not desirable to simply allocate audio vectors (arrays of sound samples) for each unit generator in the heap area, as seen in many existing computer music systems.

Considering the utilization of such embedded devices with limited memory resources for the NIME instruments, we develop a new implementation technique that can significantly reduce the memory consumption for unit-generator-based sound synthesis. While existing sound synthesis applications mostly allocate audio vectors for each unit generator in the heap area, our new implementation technique allocates audio vectors on the stack frame. Since unit generators will share and reuse these on-stack audio vectors during each DSP cycle, our technique can significantly reduce the required memory space for sound synthesis. Since such implementation allows more instruments to be played at once even with the embedded system with the scarce memory resources, it can help further research into embedded NIME systems.

2. RELATED WORK

2.1. Depth-First Traversal of a Unit Generator Graph

When computing output samples of a unit generator graph (often also called an *instrument* or a *patch*), the unit generator graph is normally traversed and computes the output samples of each unit generator in depth-first order from the output (such as the 'dac' unit generator), which is located at the root of the unit generator graph [4, p.79]. For instance, to compute the output of the unit generator graph in Figure 1, the order of the computation will be as shown in Figure 3 (in this example, the priority in depth-first traversal is given to the child node on the left side). Note that a unit generator that is visited twice or more can simply reuse the same output at the first visit and there is no need to traverse to child nodes again.

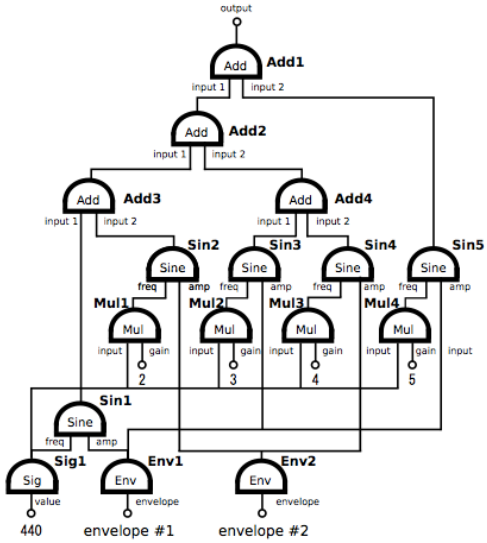


Figure 1. An additive synthesis instrument example.

2.2. Stack Machine

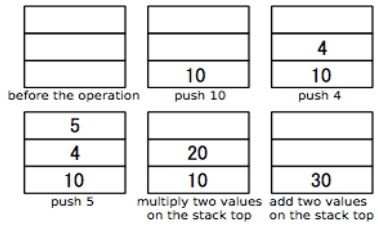


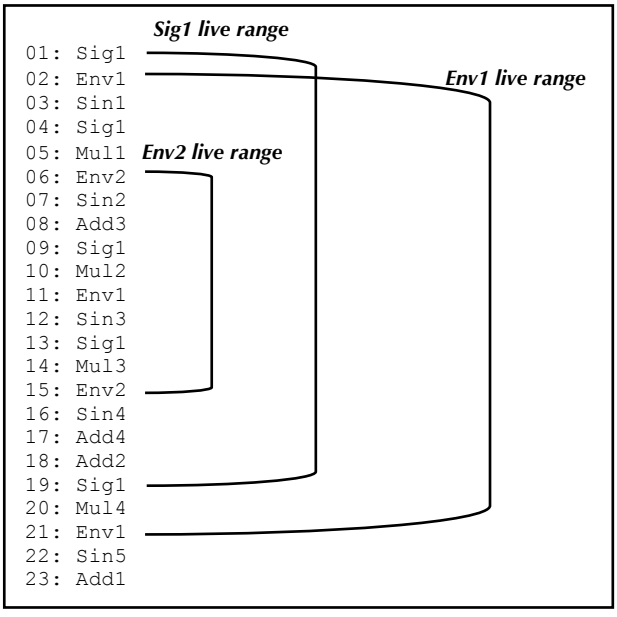
Figure 2. Calculating '10 + 4 * 5' on a stack.

A stack machine is a type of computer that utilizes a stack (a last-in first-out data structure). The Java Virtual Machine (JVM) is a typical example of a virtual machine built upon the concept of the stack machine [5]. When performing arithmetic operations on a stack machine, operands (values to be utilized) are first pushed onto its stack, and then after the operation is performed, the operands are removed from the stack and the result will be put on top of the stack. Figure 2 shows a simple example with which to compute '10 + 4 * 5' on the stack machine.

3. DESCRIPTION OF OUR TECHNIQUE

Our implementation technique applies stack-based computation during the unit-generator-based sound synthesis. The simplest implementation technique for unit generator languages/systems is to allocate audio vectors for each unit generator in the heap area and allow each unit generator to reuse its own audio vectors during sound synthesis. On the contrary, our technique allocates the arrays of audio vectors on the stack frame and allows all of the unit generators in the system to share and reuse these audio vectors on the stack. Since unit generators do not have to keep their output anymore after the last visit in each cycle (except when the output is reused in the next cycle[1]), unit generators can simply forget their output after the last use in each cycle. Hence, the same audio vectors can be reused, not only during the computation of a unit generator graph, but also for all of the unit generators. This can lead to a significant reduction of the memory resources required for sound synthesis.

In our technique, when an instrument is initially defined, we first perform depth-first traversal of a unit generator graph to obtain the order in which to compute the output of unit generators. For the unit generator graph in Figure 1, we can obtain the order in Figure 3.



[1] If the output samples must be reused in the next cycle, we simply store the output in an audio vector allocated in the heap area. Yet, only unit-generators that require such reuses over the DSP cycles needs to allocate audio vectors in the heap area; most unit-generators don't require this.

Figure 3. The order of computation in the unit generator graph and liveness analysis of the output samples of the Figure 1 instrument example.

Then, we analyze the liveness of the output to find the unit generators visited twice or more so that we can maintain and reuse their output after the first visit. The liveness analysis of the unit generator output can be performed quite simply. If the unit generator appears only once in the order, the output is not reused in the same DSP cycle. If it appears twice or more, the live range of the output is between the locations in which it is used first and last. Figure 3 shows the result of liveness analysis. As shown, three unit generators (*Sig1*, *En1*, and *Env2*) are revisited and their output must be maintained until the last use.

If any unit generator output to be reused is found, we then allocate output audio vectors for such unit generators from 0 to a larger index. In this example, the audio vectors are assigned at the indexes of 0–2 in the array of the audio vectors on the stack frame (0 for *Sig1*, 1 for *Env1*, and 2 for *Env2*). These unit generators will use the assigned audio vectors for their output. While it is not the case with this example, if the last use of such a reused audio vector is over, we can also reuse the audio vector for another reused output, if the live range is not overlapping (e.g., the last use of the reused output of a unit generator at index 0 is over, and the audio vector at the same index (0) can be used for a reused output of another audio vector).

Then, we start traversing the unit generator graph again from the root of the graph, assigning the index at the audio vector array to be used for the output of each unit generator so that they perform stack-based operation, avoiding the indexes already assigned for the reused output. In the Figure 3 example, we already assigned indexes 0–2. Hence, we begin this phase while using index 3. Since the output of these unit generators is used only once in the computation, we do not have to maintain their output audio vectors and immediately reuse the audio vectors.

```
Sin1 : 3                Mul1 : 4
Sin2 : 4                Mul2 : 4
Sin3 : 4                Mul3 : 5
Sin4 : 5                Mul4 : 4
Sin5 : 4
                        Sig1 : 0
Add1 : 3
Add2 : 3                Env1 : 1
Add3 : 3                Env2 : 2
Add4 : 4
```

Figure 4. The result of the index assignment for the Figure 1 instrument example.

Figure 4 describes the index of the audio vectors that each unit generator is assigned for its output. As shown in Figure 4, the unit generators of *Sin1*, *Sin2* and *Add3* are assigned the indexes of *3*, *4* and *3*, respectively. When computing the output of the *Add3* unit generator, it receives the pointer to the audio vectors at indexes 3 and 4. Then, it performs the addition of these two audio vectors and writes back to the audio vectors at index 3. The *Add3* unit generator utilizes the audio vector at index 3 (where the *Sin1* unit generator wrote its output) and the audio vector at index 4 (where the output of *Sin2* is written) for the input samples, and the output samples are written back to the audio vector at index 3, overwriting the output of *Sin1*.

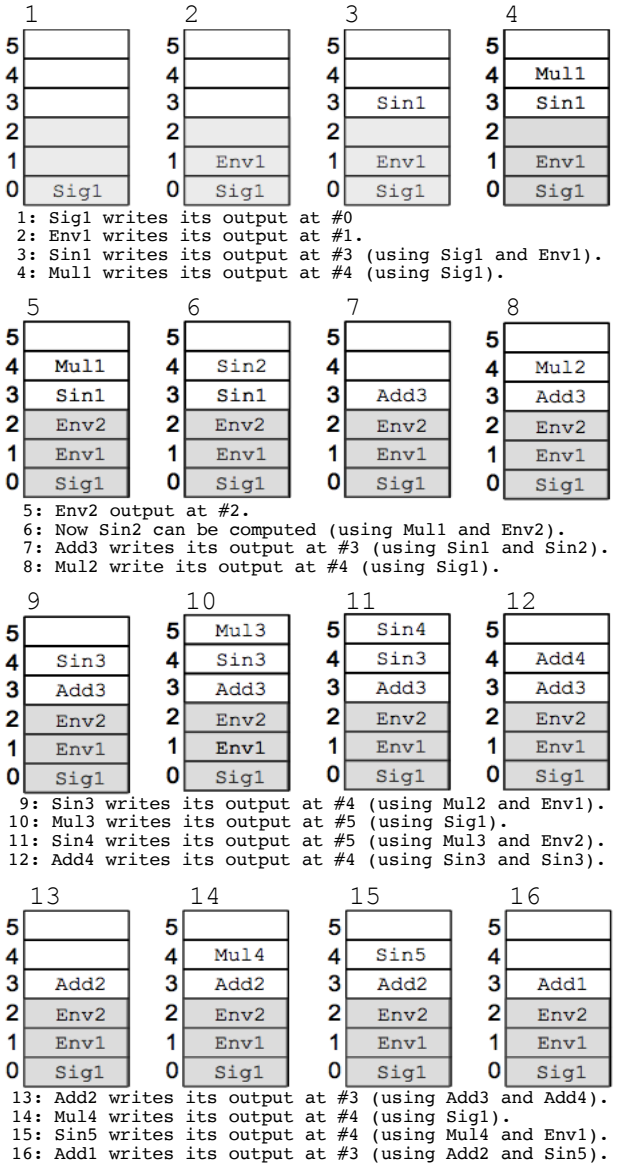


Figure 5. The process of on-stack computation in the order of the Figure 3 example.

Figure 5 illustrates how the output of the example instrument in Figure 1 can be computed on the stack

frame. The gray-shaded slots (*0*, *1*, and *2*) are used to store the reused audio vectors, and the non-shaded area is used for stack-based computation. As shown, in our stack-based computation technique, the output audio vectors from the child node(s) can be immediately reused for the output of the parent node. This contributes to reducing the size of the audio vector array. However, it is sometimes desirable not to reuse the output audio vectors of child nodes. For instance, an FIR filter may need to maintain the past input to compute its output. In this case, the object can request to assign a different audio vector to the child node during the assignment process.

Thus, by performing the computation on the stack, our implementation technique can significantly reduce the number of audio vectors utilized during the computation. As shown in Figures 4 and 5, the number of audio vectors required for the computation of the Figure 1 additive synthesis instrument example is only six. In contrast, if we utilize a simple implementation technique that allocates an audio vector to each unit generator in the heap area, we need 16 audio vectors for only one instance of the instrument.

In contrast, our technique only requires six audio vectors, regardless of the number of instances of the instrument, since the audio vectors can be reused and shared by all of the instruments. While our technique needs to preprocess to assign the index numbers of unit generators in the instrument before sound synthesis, this process is required only once when the instrument is defined or when any modification is made to the unit generator graph.

4. DISCUSSION

4.1. Memory Consumption

As mentioned in the Introduction section, our target application domain is the NIME instrument built upon the embedded device fast enough to perform real-time sound synthesis, but with scarce memory resources, which can often be seen in many embedded systems.

Suppose that we utilize the audio I/O block size of 256 samples of the type of 'float' (as in C/C++). Let us compare the required memory space between our technique and a simple implementation technique that allocates an audio vector for each unit generator. When we implement an additive sound synthesis instrument in Figure 1, if we utilize a simple implementation technique, we need 16 audio vectors for each instance of the instrument. Therefore, each instrument will need 16,384 bytes (256 samples * 4 bytes * 16 audio vectors), i.e., 16kB, for each instance. If we use eight voices at once, it consumes 128kB for the audio vectors in total. On the contrary, as shown in Figures 4 and 5, our technique requires only six audio vectors, regardless of the number of voices.

This is a quite favorable feature for our target application domain (an NIME instrument built upon an embedded device with scarce memory resources). For instance, "Arch Max v1.1" by Seeed Studio has a Cortex-M4F CPU/168MHz with an FPU (floating point unit) but is equipped with only 192kB of RAM [6]. Similarly, "Feather nRF52 Bluefruit LE" by Adafruit has a Cortex-M4F CPU/64MHz with an FPU, yet only 64kB of SRAM is available [7]. Note that the other non-sound synthesis parts of a user program also require some memory space; the significant reduction of memory consumption in our technique is quite beneficial for this target application domain.

4.2. Memory Allocation

Memory allocation on the stack frame can be performed very quickly in comparison with memory allocation from the heap area, since the allocation on the stack frame only requires moving the stack pointer appropriately (the 'alloca' function can be utilized in the C programming language for this purpose). Even though the memory allocated on the stack frame is gone when the function is exited, this does not matter for our implementation technique, as it does not require unit generators to hold the memory space until the next DSP cycle.

Moreover, it should be noted that the reduction of the required memory resources can also help to lower the time cost for the instantiation of an instrument object. As the CPU time cost for the memory allocation from the heap area (such as the 'malloc' function in the C programming language) can be large, computer music systems normally pool some memory beforehand to avoid the time cost for the heap memory allocation during the instantiation of an instrument object.

Yet, since our technique can reduce the entire memory resources required for unit generators, a computer music system can reduce the entire size of the memory pool in the heap area, which is used for unit generators. This characteristic is also quite favorable in achieving efficiency in both memory consumption and computational performance at once.

4.3. Performance Efficiency

Some might consider the computational efficiency to be improved, since our technique reuses the same memory space on the stack fame, resulting in fewer CPU cache misses. We investigated this issue on a personal computer with a CPU with L2 and L3 caches[2]. We used

[2] The test was performed on the Apple Mac Book Air (11-inch, Early 2015, Intel Core i5 1.6GHz, 4GB Memory, Mac OS X El Captain).

the Intel Performance Counter Monitor [8] and tested simple sound synthesis tasks such as additive synthesis and FM synthesis as a preliminary study. While we observed a good improvement in cache hits/misses (as expected), improvement in the entire performance efficiency was not observed in our experiment.

While further investigation is required, one of the possible hypotheses would be that prefetching from memory to the L3 cache may play a significant role in unit-generator-based synthesis. If unit generators and audio vectors are allocated side by side in memory, prefetching can be effectively performed. As the computation of the output samples from a unit generator can take some execution time, the memory space allocated to the unit generator that is computed next in the order of depth-first traversal may be already available on the CPU cache by prefetching when it is computed.

4.4. Lazily Evaluated Audio Vectors

Our previous research work proposed a new implementation technique that utilizes lazy evaluation of audio vectors for unit-generator-based sound synthesis [9]. This technique is beneficial for minimizing the audio I/O latency and for stabilizing the behavior in digital sound synthesis, and would also be potentially useful for parallelization, distribution and speculation in unit-generator-based digital sound synthesis.

However, since a unit generator graph produces the tree of lazily evaluated audio vectors in every DSP cycle in this implementation technique, the current implementation described in [9] pools audio vectors within the unit generators beforehand. This can significantly increase the size of the memory pool that must be prepared, and may cause some unneglectable overhead for the runtime performance efficiency in the worst case.

As we already discussed in [9], this on-stack computation technique of audio vectors can be combined with the implementation technique that utilizes lazily evaluated audio vectors, so that the required memory size for the lazy evaluation of audio vectors can be minimized. This can lead to the possibility of adopting the lazily evaluated audio vectors in an embedded NIME system, which may minimize the audio I/O latency and improve the stability of the system at once.

5. CONCLUSION AND FUTURE WORK

We described our novel implementation technique, i.e., on-stack computation of audio vectors for unit-generator-based sound synthesis. Our technique can significantly reduce the required memory space and such a feature can benefit further research into the embedded NIME instrument. We are currently planning to improve our technique further and to develop a software framework that utilizes our technique for sound synthesis on embedded devices. The Cortex-M4F devices are of our particular interest, as the CPU is equipped with an FPU, which is a desirable feature for digital sound synthesis and digital sound processing.

We also discussed how this stack-based computation technique of audio vectors may be beneficial in reducing memory consumption and related overhead in the implementation technique of lazily evaluated audio vectors that we previously developed, as it can remove the object pool of audio vectors from the implementation technique. This can help to extend the application domain of the implementation technique to embedded NIME systems. We also plan to combine our stack-based computation technique with the implementation technique of the lazily evaluated audio vectors.

6. REFERENCES

[1] Berdahl, E. and Wendy, J., "Satellite CCRMA: A Musical Interaction and Sound Synthesis Platform," *Proc. of Nime,* 2011.

[2] Richardson, M. and Wallace, S., *Getting Started with Raspberry Pi*, O'Reilly Media Inc., 2012.

[3] Toulson, R. and Wilmshurst, T., *Fast and Effective Embedded Systems Design: Applying the ARM mbed*, Newnes, 2016.

[4] Wang, G., *The ChucK Audio Programming Language: A Strongly-timed and On-the-fly Environ/mentality*, Princeton University, 2008.

[5] Lindholm, T. et al., *The Java Virtual Machine Specification*, Pearson Education, 2014.

[6] Seeed Studio, *Arch_Max_v1.1 – Seeed Wiki,* http://wiki.seeed.cc/Arch_Max_v1.1/ (accessed on Aug/31/2018).

[7] Adafruit, *Introduction | Bluefruit nRF52 Feather Learning Guide | Adafruit Learning System*, https://learn.adafruit.com/bluefruit-nrf52-feather-learning-guide?view=all (accessed on Aug/31/2018).

[8] Willhalm, T. et al., "Intel Performance Counter Monitor – A Better Way to Measure CPU Utilization," https://software.intel.com/en-us/articles/intel-performance-counter-monitor (accessed on Aug/31/2018).

[9] Nishino, H., "Unit-generator Graph as a Generator of Lazily Evaluated Audio-vector Trees," *Proc. of Sound and Music Computing*, 2018.