CONSERVATIVE GARBAGE COLLECTORS FOR REALTIME AUDIO PROCESSING

Kjetil Matheussen

Norwegian Center for Technology in Music and the Arts. (NOTAM) *

k.s.matheussen@notam02.no

ABSTRACT

Garbage-producing and efficient programming languages such as Haskell, Lisp or ML have traditionally not been used for generating individual samples in realtime. The reason is a lack of garbage collector fast and predictably enough to make these languages viable alternatives to C and C++ for high performing audio DSP. This paper shows how conservative garbage collectors can be used efficiently for realtime audio signal processing.

Two new garbage collectors are described. The first collector can easily replace garbage collectors in many existing programming languages and has successfully been used for the Stalin Scheme implementation. The second garbage collector has a higher memory overhead and requires a simple write barrier, but does not restrain the amount of memory.

Both collectors find garbage by running a parallel mark-and-sweep on snapshots. The snapshot buffers are either copied to between soundcard interrupts or filled up by write barriers. To ensure predictability, worst-case is simulated about once a second, and all running garbage collectors are synchronized to avoid more than one garbage collector to block audio simultaneously. High performance is maintained since the collectors should not interfere with inner audio loops. Benchmarks also show that overhead can be minimal.

1. INTRODUCTION

It is common for music programming systems to provide pre-programmed sound generators which process blocks of samples at a time, called "block processing". Block processing makes it possible to achieve high sample throughput without using low-level languages such as C or C++. Controlling the graph of generators either happens in a separate process (SuperCollider3 [8]), or by performing smaller jobs in between computing blocks (Pure Data [9]).

However, only being able to process blocks of samples is sometimes limiting. This paper describes two garbage collectors supporting hard realtime computation of individual samples in high-level languages such as Haskell, Lisp or ML.

Some relevant properties for realtime audio processing:

1. The only deadline is the next soundcard interrupt, meaning that it's only important to compute sound fast enough. Events generated by the keyboard or other sources do not require other immediate response than the change in sound they might cause, and therefore these events can be recorded in an external process not requiring garbage collection. The audio process runs at regular intervals, processing blocks of samples, and handles the events recorded by the external process since last time. To achieve perfect accuracy, the events can be timestamped.
2. Audio processing algorithms are usually small and require relatively little pointer-containing memory (most allocated memory is usually sample buffers). Moreover, in music programming systems, it is common to run many smaller jobs simultaneously. These jobs are commonly called "instruments". The amount of memory per heap is reduced if each instrument uses its own heap.
3. Audio often runs on a general computer where it's not known beforehand how many programs are running at once or how they may use a garbage collector. Therefore, excessive CPU usage can stack up in unpredictable ways between soundcard interrupts. One way to avoid this is to let all programs have strict constant CPU usage, where *best-case* is always equal to *worst-case*. Another way is to synchronize programs so that only one program uses excessive CPU at a time.
4. Audio often involves interaction with a user, either by writing code which performs music directly [3], or by making applications which depend on user input. Either way, it is usually not possible to interact very well with programs if the computer uses more than about 80% CPU time, since GUI processes, keyboard input, etc. do not respond very fast then. This leaves about 20% CPU time, which can be used occasionally by audio code without increasing the chance of glitches in sound or destroy interactivity.
5. Realtime garbage collectors commonly guarantee a fixed number of milliseconds of worst-case execution time. But for audio processing, it may also be necessary for the user to immediately (i.e. within a second or so) know the consequence of the worst-case. If not, unpredictable sound glitches may occur.

For example, if a DSP code uses a constant 80% CPU, there

* Also; Department of Informatics, University of Oslo.

will be a glitch in sound if the collector uses more than 20% CPU within one audio block. If using more than 20% CPU only about once an hour, the user would have to test at least an hour to be comfortable that the code runs properly.

Not being immediately sure whether there will be enough CPU is unfortunate since music programming is often experimental, where the user doesn't want to think too much about whether the code will always run properly. And when performing live, perhaps even writing code on stage, it's not always an option to test first. And furthermore, as per point 3, it becomes harder to know how CPU usage between simultaneously running programs may add up if some of the programs have a non-immediate worst-case.

1.1. Conservative Garbage Collectors

Both collectors described in this paper are conservative. A conservative garbage collector considers all memory positions in the root set or heap as potentially containing a pointer. When considering all memory positions as potentially holding a pointer, the code interfacing the collector becomes simpler since it can avoid defining exact pointer positions, which also makes it easier to replace garbage collectors in already existing language implementations. For audio DSP, this also means that inner audio loops should run unmodified.

Many language implementations, such as Stalin Scheme, Bigloo Scheme, D, and Mono are using a conservative garbage collector, very often the Boehm-Demers-Weiser garbage collector (BDW-GC) [4]. And since the garbage collector often is the biggest (perhaps also the only) hindrance for realtime usage, it becomes relatively simple to enable these languages to produce audio reliably in realtime.

It is however questionable whether a conservative collector can guarantee hard realtime behavior. There are two reasons why it might not: (1) Fragmentation can cause programs to run out of memory prematurely. (2) Values in memory misinterpreted as pointers (false pointers) can cause unreferenced memory not to be reclaimed. However, false pointers is unlikely to be a problem on machines with 32 bit or higher address space, and fragmentation can be prevented from becoming a problem by using a safety buffer.

2. A SIMPLE CONSERVATIVE GARBAGE COLLECTOR FOR AUDIO PROCESSING

There are two basic ideas:

1. **Simulating worst-case**
   To achieve a constant execution time and a predictable overhead, the collector is forced to spend worst-case amount of time between blocks.
2. **Finding garbage in a parallel thread**
   By finding garbage in a parallel lower-priority thread, the only critical operation is preparing for a parallel garbage collection. Simulating worst-case only by preparing a parallel collection can be both reliable and simple.

2.1. The Basic Technique

The collector works by first allocating a fixed size heap small enough to be fully copied within this time frame:

$$m - s \tag{1}$$

where m is the duration of one audio buffer, and s is the time a program uses to process all samples in that block of audio. This heap is used for storing pointer-containing memory. To achieve a consistent CPU usage during the lifetime of a program, its size can not be increased.

'$m-s$' means that the size of the pointer-containing heap is restricted by the speed of the computer and audio latency. However, since audio data do not contain pointers, allocation of for instance delay lines or FFT data is not restricted. Such data can instead be allocated from a separate heap which does not have to be copied.

Copying the pointer-containing heap within the '$m-s$' time frame is called "taking snapshot". After taking a snapshot, an independently running lower-priority thread is signaled. The lower-priority thread then finds garbage by running a mark-and-sweep on the snapshot.

Program 1 Basic version

```
 1  mark-and-sweep thread()
 2    loop forever
 3      wait for mark-and-sweep semaphore
 4      run mark and sweep on snapshot
 5
 6  audio function()
 7    produce audio
 8    if mark-and-sweep is waiting then
 9       copy heap_pointers and roots to snapshot
10       if there might be garbage then
11          signal mark-and-sweep semaphore
12       endif
13    else
14       copy heap_pointers and roots to a dummy-snapshot
15    endif
```

Program 1 shows what has been described so far. (Keep in mind that Program 1 is only ment as a simple overview of the basic technique. It has a few shortcomings and problems which are addressed later in this paper.)

Some comments on Program 1:

- "*audio function()*" on line 6 is called at regular intervals to process one block of audio.
- On line 14, the heap is copied to a "*dummy-snapshot*". Copying to a dummy snapshot is necessary to ensure a constant overhead and to avoid invalidating the real snapshot while the garbage collector is running.

  Alternatively, we could tell the operating system to sleep instead of copying to a dummy snapshot, so that other programs could run in the mean time. However, to avoid waiting too long or too short, the sleeping function would need to provide sub-ms accuracy, which is not always available.
- The check for "*if there might be garbage*" could for instance be performed by checking the amount of allocated memory since last collection. This check is not required for correct operation, but lowers overall CPU usage.

- The "*mark-and-sweep*" thread (our "lower-priority thread") should run with a lower priority than the audio thread so that it won't steal time from the audio function. But, this thread still needs to run with realtime priority or a very high priority, so that GUI updates etc. can not delay memory from being reclaimed.

If instruments or applications depend on each other, for example if one instrument produces audio used by a subsequent reverberation instrument, performance can increase if the snapshot is taken in parallel, as shown in Program 2.

Program 2 Parallel snapshot

```
mark-and-sweep thread()
  loop forever
    wait for mark-and-sweep semaphore
    run mark and sweep on snapshot

snapshot thread()
  loop forever
    wait for snapshot semaphore
    if mark-and-sweep is waiting then
       copy heap_pointers and roots to snapshot
       if there might be garbage then
          signal mark-and-sweep semaphore
    else
       copy heap_pointers and roots to dummy-snapshot
    endif
    signal audio function semaphore

audio function()
  wait for audio function semaphore
  produce audio
  signal snapshot semaphore
```

2.2. Memory Overhead

The size of the snapshot and the dummy-snapshot is equal to the heap. When running only one instrument, which requires its own snapshot and dummy snapshot, the memory usage will triple. But, when several instruments uses the same garbage collector, where each of them have its own heap, and only one instrument runs a garbage collection at a time, the overhead becomes

$$\frac{n+2}{n} \tag{2}$$

where n is the number of instruments.

3. SYNCHRONIZING GARBAGE COLLECTORS

The solution proposed so far is not ideal. Firstly, it's not obvious how to create an algorithm to get consistant execution times for taking snapshots if more than one garbage collector is running at the same time. Secondly, taking full snapshots between the processing of every audio buffer wastes a lot of CPU since it's not always useful to collect garbage that often, plus that only parts of the heap is normally used. Thirdly, the very CPU-intensive use of a dummy snapshot serves no other purpose than timing. To avoid these problems, all simultaneously running garbage collectors must be synchronized. By making sure only one snapshot is taken at a time on the computer, excessive CPU usage does not stack up, and it becomes possible to reduce CPU usage in manifold ways.

3.1. Non-Constant Overhead

By synchronizing garbage collectors (or by running only *one* garbage collector at a time), the following optimizations can be applied:

1. It is only necessary to take snapshot of the complete heap about once a second (called "full snapshot"). In between, only the used part(s) of the heap can be copied instead (called "partial snapshot"). Since the soundcard buffer is normally refilled somewhere between 50-1500 times a second, this will eliminate most of the wasted CPU. The user will still notice when the garbage collector spends too much time, but now only once a second, which should be good enough.
2. By making sure that snapshots are never taken two blocks in a row, spare-time will be available both after producing audio in the current block, and before producing audio in the next. The time available for taking snapshots will now be:
$$2*(m-s) \tag{3}$$
where m is the duration of one audio buffer, and s is the duration of processing the samples in that buffer.
3. In case producing audio for any reason takes longer time than usual, which for instance can happen if creating sound generators or initializing sample buffers, worst-case can be lowered by delaying a new snapshot until the next block. In case a snapshot on a heap not belonging to the current program has already been initiated in the current block cycle, that snapshot (plus its mark-and-sweep) can be cancelled and the corresponding audio function can be told to run without having to wait for the snapshot to finish.

An implementation of a "one-instance only" garbage collector is shown in Program 3. Program 3 differs from one being synchronized by how it decides whether to signal a new garbage collection, and whether to do a full snapshot.

3.2. The Synchronization

There are two basic ways to synchronize garbage collectors. The first way is to let the server select which client is allowed to run a garbage collection in the current audio buffer cycle. The second way is to let the server itself do the garbage collection.

Advantages of letting the server itself do the garbage collection are: (1) Lower memory overhead: Only one snapshot buffer is needed on the computer. (2) Only one mark-and-sweep process is run simultaneously: Less code and memory is used, which is better for the CPU cache. (3) The garbage collector runs in a different memory environment: Making code simpler by automatically removing any chance of false sharing.

One disadvantage of running the garbage collection on a server is that it may have to use a large amount of shared memory. To ensure high performance, shared memory will

be required both for the snapshot heap, roots, and perhaps for communication between server and clients. Since shared memory is often a limited resource, this could be a problem.

Program 3 Non-constant overhead

```
mark-and-sweep thread()
  loop forever
    wait for mark-and-sweep semaphore
    run mark and sweep on snapshot

snapshot thread()
  loop forever
    wait for snapshot semaphore
    if at least one second since last full snapshot then
       copy roots to snapshot
       copy full heap_pointers to snapshot
       if there might be garbage then
          signal mark-and-sweep
    else if there might be garbage then
       copy roots to snapshot
       copy partial heap_pointers to snapshot
       signal mark-and-sweep
    else
       do nothing
    endif
    signal audio function

audio function()
  wait for audio function semaphore
  produce audio
  if snapshot is waiting, and
     mark-and-sweep is waiting, and
     no snapshot was performed last time, and
     it didn't take a long time to produce audio
  then
     signal snapshot
  else
     signal audio function
  endif
```

3.3. Memory Overhead

Since the garbage collector does not need to provide consistent CPU overhead when being synchronized, the dummy snapshot is not needed anymore. Therefore the memory overhead now becomes:

$$\frac{n+1}{n} \tag{4}$$

where n is the number of heaps connected to or used by the garbage collector. [1]

4. IMPLEMENTATION OF ROLLENDURCHMESSERZEITSAMMLER

Rollendurchmesserzeitsammler[2] (Rollendurch) is a freely available garbage collector for C and C++ which uses the techniques described so far in this paper. Rollendurch is also made to replace the BDW-GC collector [4] in existing language implementations. Although some of the more advanced features in BDW-GC are missing, Rollendurch is still successfully being used instead of BDW-GC in C sources created by the Stalin Scheme compiler [10, 7].

[1] However, if running garbage collection on a server, it could be necessary to add another snapshot plus another set of garbage collector threads to avoid sometimes running both the audio thread and the garbage collector on the same CPU or access the same snapshot from two different CPUs. In that case, the memory overhead will be $\frac{n+2}{n}$.

[2] http://users.notam02.no/~kjetism/rollendurchmesserzeitsammler/

4.1. Implementation Details

Achieving perfect realtime performance with modern general computers is impossible. The reason is unpredictable thread scheduling schemes, CPU caches, multiple processors and other factors caused by operating system and hardware. Therefore, Rollendurch adds some extra logic to avoid unpredictable execution times:

- After allocating memory, a pointer to the allocated memory block and its size are transported on a ringbuffer from the audio thread to the garbage collector threads. No information about the heap used by the garbage collector, such as pointers to the next allocated memory block, is stored in the heap itself. This not only prevents the garbage collector thread from having to access the heap (which could generate unpredictable cache misses), but it also makes more memory available in the heap and lowers snapshot time.
- Similarly, in order for *sweep* to avoid calling the free function in the memory manager when finding unreferenced memory, sweep instead sends sizes and addresses on a ringbuffer to a special "free thread". The "free thread" takes care of freeing memory and it is specified to run on the same CPU as the audio thread, hence avoiding cache misses caused by accessing the heap from another CPU.
- Since a conservative garbage collector is not always using the same amount of memory every time a program is run, and especially not in one where the garbage collection runs in parallel with threads allocating memory, 3/4 of the heap in Rollendurch is dedicated as a safety buffer. If a program spends more than 1/4 of the heap, a window will appear on the screen warning the user that the collector can not guarantee hard realtime performance anymore. For example, in a concert situation, if the program suddenly uses slightly more than 1/4 of the heap (which can happen even if the program had been previously tested not to use more than 1/4), the performance of the garbage collector would still be essentially the same and the collector would not contribute to glitches in sound.

  Only guaranteeing hard realtime performance for 1/4 of the heap (and slightly above) also makes it possible to accurately adjust the time it takes to simulate worst-case. In Rollendurch, simulating worst-case happens by first copying 1/4 of the heap (or more in case more than 1/4 of the heap is actually in use), and then continue copying the remaining heap in small parts as far as possible until a predefined snapshot duration is reached. In other words, taking full snapshot also serves as a busy-loop timer. The predefined snapshot duration is calculated at program initialization by running several full snapshots in a row and returning the smallest duration of those. The ratio 1/4 was selected based on observed variance in time taking snapshot.
- Although they have almost the same API, the mentioned ringbuffers are not really ringbuffers, but a data structure

containing two stacks: One stack is used for reading and the other for writing. The two stacks switch position at the next garbage collection. Cache misses caused by transporting information about allocated memory back and forth between threads running on two different CPUs are now limited to one occurrence per memory position between each new garbage collection.

4.2. Memory Management

Program 4 shows the default memory manager. Since the heaps can become extremely fragmented with this memory manager, Rollendurch also provides an option to use the "Two Level Segregated Fit" memory manager (TLSF) [6] instead. TLSF is a memory manager made for hard realtime operation which handles fragmentation much better, but TLSF also spends a little bit more time allocating and deallocating.

Program 4 The default memory manager

```
alloc(size)
  if pools[size] != NULL then
     return pop(pools[size])
  else
     freemem += size
     return freemem-size
  endif

free(mem, size)
  if freemem-size == mem then
     freemem -= size
  else
     push(mem, pools[size])
  endif
```

4.3. Benchmarks

In these benchmarks, the results of Rollendurch is compared with the results of a traditional mark-and-sweep collector. The mark-and-sweep collector was made by slightly modifying Rollendurch. (Compile time option: NO_SNAPSHOT). Both Rollendurch and the traditional mark-and-sweep collector used the efficient memory manager in Program 4. Furthermore, both garbage collectors also used large hash tables for storing pointers to allocated memory, which should lower the blocking time for the mark-and-sweep collector.

Program 5 Minimal MIDI player written for Stalin Scheme

```
(<rt-stalin> :runtime-checks #f
 (while #t
   (wait-midi :command note-on
     (define phase 0.0)
     (define phase-inc (hz->radians (midi->hz (midi-note))))
     (define tone (sound
                    (out (* (midi-vol) (sin phase)))
                    (inc! phase phase-inc)))
     (spawn
       (wait-midi :command note-off :note (midi-note)
         (stop tone))))))
```

For the tests, a 3.10 minute long MIDI file[3] was played using Program 5. Program 5 is a Stalin Scheme program running in the Snd-Rt music programming system [7]. The processor was a dual-core Intel T5600 processor (1833MHz) with a shared 2MB of L2 cache running in 32 bit mode. The computer used DDR2 memory running at 667MHz. The T5600 processor does not have an integrated memory controller, which could have significantly increased the snapshot performance. Time values were gathered using the CPU time stamp counter (TSC), and all threads were running in realtime using either the SCHED_FIFO or the SCHED_RR scheduling scheme. All threads were also locked to run on one CPU only. The size of the pointer-containing heap was set to 1MB. At most 2.8kB was allocated by the program at any point in time, and a constant 8kB is allocated during initialization by Snd-Rt for storing pointers to coroutines. (Rollendurch should provide predictable performance up to 256kB when using a 1MB pointer-containing heap.) The size of the non-pointer-containing heap was set to 4MB, but the size of the non-pointer-containing heap should not affect the performance in any way.

Mark-and-sweep			Rollendurch		
Min	Avg.	Max	Min	Avg.	Max
0.00ms	0.03ms	0.16ms	0.02ms	0.03ms	0.10ms

Table 1. Blocking time during one audio buffer.

Table 1 shows the amount of time the garbage collector blocks audio processing during one audio buffer. (For Rollendurch, this is the time taking partial snapshot and root snapshot, while for mark-and-sweep it is the time running *mark*). Note that 80.5% (87.4ms out of 108.7ms) of the time spent by Rollendurch was used taking snapshot of the roots, which is a constant cost. Furthermore, Rollendurch takes snapshot of the roots even when it is not necessary, just to keep the CPU cache warm, while mark-and-sweep never have to take snapshot of the roots.

Mark-and-sweep			Rollendurch		
Min	Avg.	Max	Min	Avg.	Max
n/a	n/a	n/a	0.42ms	0.43ms	0.47ms

Table 2. Time simulating worst-case.

Table 2 shows the amount of time the collector blocks audio processing during one audio buffer when simulating worst-case. Simulating worst-case is required to achieve a predictable performance and happens about once a second. Simulating worst-case for mark-and-sweep has not been implemented since it is not known how one would do that. But if comparing the total time taking partial snapshot in Rollendurch by the total time running *mark* in mark-and-sweep ($\frac{123.5ms}{21.2ms}$), worst-case for mark-and-sweep would be about 5.8 times higher than Rollendurch. Comparing worst-case occurrences (instead of combined time) indicates that worst-case for mark-and-sweep is about 6.5 times higher than Rollendurch ($\frac{0.156ms}{0.0240ms}$), but these numbers are more prone to measurement errors than total time. Worst-case could also be higher for mark-and-sweep since it's unknown how much extra time may be spent handling interior pointers (pointers

[3] http://www.midisite.co.uk/midi_search/malaguena.html

pointing inside a memory block), and how much extra memory may be unnecessarily scanned because of false pointers. However, it is problematic to base the the performance of worst-case on the performance of non-worst-case since worst-case is more likely to use memory not in the CPU cache. Furthermore, mark-and-sweep probably spent a large amount of time scanning roots, making it even harder to predict a worst-case.

Mark-and-sweep	Rollendurch
3.15% (0.07%)	3.22% (0.16%)

Table 3. CPU usage to play the song.

Table 3 shows average CPU usage of the program to play the song. The numbers in the parenthesis show the separate CPU usage of the two garbage collectors only. Rollendurch used a bit more than twice as much CPU than Mark-and-sweep, but 36.67% of Rollendurch ran in parallel. The amount of time running in parallel would have been higher if the program had used more memory. The program using the mark-and-sweep collector also spent 46 more milliseconds to calculate the samples itself, which might be because mark-and-sweep ran on the same CPU as the audio thread and therefore could have cooled down the L1 cache.

The time data generated by running the benchmarks can be downloaded from `http://users.notam02.no/~kjetism/rollendurchmesserzeitsammler/icmc2009/`

5. A GARBAGE COLLECTOR FOR AUDIO PROCESSING, NOW WITH A WRITE BARRIER

There are a couple of limitations with the first garbage collector:

1. Maximum heap size depends directly on the speed of the computer and audio buffer size. If the audio buffer size is reduced by two, the time available for taking a snapshot is reduced by two as well.
2. If running out of memory, and the programmer increases the size of the heap, then a larger part of the available time within one audio buffer is spent taking snapshot, and the time available for doing audio processing is reduced. To avoid pre-allocating too much memory for safety, and not waste CPU taking snapshot of the safety memory, it would be preferable if the collector could monitor the heap and automatically allocate more memory when there's little left, but then the CPU usage would be unpredictable.

This section presents a solution to these problems using a very simple write barrier plus increasing the memory overhead. In this collector, taking a snapshot of the roots is the only added cost between audio interrupts, which should normally be a very light operation and also independent of heap size. Furthermore, if memory is low, additional memory can be added to the collector during runtime without also increasing worst-case (although the write barrier in Program 7 needs to be extended a little bit first).

5.1. Implementation

To make the write barrier as efficient as possible, two extra memory blocks are used to store newly written heap-pointers. These two memory blocks have the same size as the heap and the snapshot. The write barrier writes to one of these memory blocks (*memblock_wb*), while the garbage collector unloads pointers from the other (*memblock_gc*). The two memory blocks swap position before initiating a new garbage collection.

Using this write barrier, writing a pointer to the heap only requires two or three times as many instructions as before. (In addition, a couple of extra instructions are required if it's necessary to check whether the target address belongs to the heap.)

Program 6 The second garbage collector

```
ps                    = sizeof(void*)

heap_pointers         = calloc(ps,FULL_HEAP_SIZE)

snapshot              = calloc(ps,FULL_HEAP_SIZE)
memblock_wb           = calloc(ps,FULL_HEAP_SIZE)
memblock_gc           = calloc(ps,FULL_HEAP_SIZE)

snap_offset           = memblock_gc - heap_pointers

roots_snapshot        = malloc(MAX_ROOT_SIZE)
dummy_roots_snapshot  = malloc(MAX_ROOT_SIZE)

UNUSED                = -1 // Could be NULL as well...

init()
  start_lower_priority_thread(gc_thread)
  for i = 0 to FULL_HEAP_SIZE do
      memblock_gc[i] = UNUSED
      memblock_wb[i] = UNUSED

unload_and_reset_memblock()
  for i = 0 to current_size(heap_pointers) do
      if memblock_gc[i] != UNUSED then
         snapshot[i]     = memblock_gc[i]
         memblock_gc[i]  = UNUSED
      endif

gc_thread()
  loop forever
    wait for collector semaphore
    unload_and_reset_memblock()
    run_mark_and_sweep on snapshot

audio function()
  produce audio
  if collector is waiting and there might be garbage then
     swap(&memblock_gc,&memblock_wb)
     snap_offset = memblock_wb - heap_pointers
     copy roots to roots_snapshot
     signal collector
  else
     copy roots to dummy_roots_snapshot
  endif
```

Program 7 The write barrier implemented in C

```
#define write_heap_pointer(MEMPOS,POINTER)do{        \
  void* _gc_pointer=(void*)POINTER;                  \
  char* _gc_mempos=(char*)MEMPOS;                    \
  *((void**)(_gc_mempos+snap_offset)) = _gc_pointer; \
  *((void**)_gc_mempos)               = _gc_pointer; \
}while(0)
```

5.2. But what about the Cache?

Although writing pointers to the heap shouldn't normally happen inside inner audio loops, hence the additional number of instructions executed by the write barrier should be low, performance could still be reduced since switching between two memory blocks increases the chance of cache misses. This is especially unfortunate in case "gc_thread" runs on a different CPU (to increase general performance) since memory then has to be transported from one cache to another.

A solution to the problem is using another thread for resetting *memblock_gc*, one which is specified to run on the same CPU as the audio function. By using a dedicated reset thread, the garbage collector will never write to *memblock_wb* or *memblock_gc*, and no memory block is written to by more than one CPU. This scheme is implemented in Program 8.

Unfortunately, switching between two memory blocks can still cause cache misses, even on a single CPU. And furthermore, the number of cache misses can increase if one memory block hasn't been used by the audio code for a very long time. Program 8 also lowers the chance of unpredictable cache misses by switching between memory blocks as often as possible.

Another way to swap memory blocks more often is to run mark-and-sweep in its own thread, and delay the unloading of newly written pointers by putting them on a temporary buffer. Before initiating a new mark-and-sweep, the pointers on the temporary buffer are copied into the snapshot, and the buffer is marked as clear. In case the temporary buffer is full when unloading pointers, the garbage collector thread just waits until the mark-and-sweep thread is finished, and then writes to the snapshot directly. When writing to a temporary buffer instead, the memory blocks can be switched even while mark-and-sweep is running. This scheme is not implemented in Program 8.

However, swapping memory blocks frequently might not be good enough, so future work is benchmarking other alternatives such as using a ringbuffer, a hash table, or a couple of stacks to store newly written heap-pointers.

5.3. Memory Overhead

In this collector, the snapshot can not be shared between several instruments since the content of the snapshot in the previous collection is also being used in the current. One of the memory blocks can however be shared since it's only used while initiating a new garbage collection, so the overhead now becomes

$$\frac{n*3+1}{n} \tag{5}$$

where n is the number of heaps connected to or used by the garbage collector.

5.4. Write Barrier for the Roots

It is possible to avoid taking snapshot of the roots as well if we also use write barriers when writing pointers to the roots. However, since taking a snapshot of the roots shouldn't normally take much time, and that the extra number of write barriers higher the risk for some of them to occur inside inner audio loops, overall performance could significantly decrease.

On the other hand, by including the program stack in the root set, a new garbage collection could be initiated anywhere in code and take virtually no time, making the collector capable of hard realtime also for other types of use.

Program 8 A more cache-friendly version

```
init()
  start_lower_priority_thread(reset_memblock_thread)
  start_lower_priority_thread(gc_thread)

  set_cpu_affinity(0,reset_thread)
  set_cpu_affinity(0,audio_thread)
  set_cpu_affinity(1,gc_thread)

  for i = 0 to FULL_HEAP_SIZE do
      memblock_gc[i] = UNUSED
      memblock_wb[i] = UNUSED

reset_memblock_thread()
  loop forever
    wait for reset semaphore
    for i = 0 to current_size(heap_pointers) do
        if memblock_gc[i] != UNUSED then
           memblock_gc[i]  = UNUSED
        endif

unload_memblock()
  for i = 0 to current_size(heap_pointers) do
      if memblock_gc[i] != UNUSED then
         snapshot[i] = memblock_gc[i]
      endif

gc_thread()
  loop forever
    wait for collector semaphore
    unload_memblock()
    signal reset semaphore
    if there might be garbage then
       run_mark_and_sweep on snapshot
    endif

audio function()
  produce audio
  if first time then
     do nothing
  else if the dummy snapshot was not used last time then
     copy roots to dummy_roots_snapshot
  else if collector is waiting, and reset is waiting, then
     swap(&memblock_gc,&memblock_wb)
     snap_offset = memblock_wb - heap_pointers
     copy roots to roots_snapshot
     signal collector
  else if mark-and-sweep is using roots_snapshot then
     copy roots to dummy_roots_snapshot
  else
     copy roots to roots_snapshot
  endif
```

6. COMPARISON WITH EARLIER WORK

The author is not aware of any existing work on garbage collectors specifically made for audio processing, but a few general realtime garbage collectors have been used for audio earlier.

James McCartney's music programming system Supercollider3 [8] implements the Johnstone-Wilson realtime collector[4] [5] to handle input events and control a sound graph. However, the garbage collector in Supercollider3 is not used by the program producing samples, only by the program controlling the sound graph.

Vessel [11] is a system running running on top of the Lua programming language (http://www.lua.org). Lua has an incremental garbage collector, and contrary to Supercollider3, this collector runs in the same process as the one generating samples. It might also be possible to edit individual samples in realtime in Vessel.

The Metronome garbage collector [2, 1] provides an interface to define how much time can be spent by the collector within specified time intervals, hence the name "Metronome". This interface makes it possible to configure the

[4] This according to an Internet post by James McCartney. (`http://lambda-the-ultimate.org/node/2393`)

Metronome collector to use a specified amount of time in between soundcard interrupts, similar to the collectors described in this paper. However, the Metronome collector is only available for Java, and it requires a read barrier. It is also uncertain how consistent the CPU usage of this collector is. For instance, in [1] it is said that:

> "an MMU of 70% with a 10 ms time window means that for any 10 ms time period in the program's execution, the application will receive at least 70% of the CPU time (i.e., at least 7 ms)"

The collectors described in this paper either guarantee a constant overall CPU time, or a guarantee that the user immediately will discover the consequence of only getting a certain amount of CPU. Furthermore, the Metronome collector does not seem to synchronize simultaneously running collectors to avoid worst-case behaviors to stack up unpredictably.

7. CONCLUSION

This paper has presented two garbage collectors especially made for high performing realtime audio signal processing. Both garbage collectors address two properties: 1. The user needs to immediately know the consequence of worst-case execution time. 2. Several applications may run simultaneously, all sharing common resources and producing samples within the same time intervals.

The first garbage collector can relatively easily replace garbage collectors in many existing programming languages, and has successfully been used for the Stalin Scheme implementation. The second garbage collector requires at least twice as much memory, and a very simple write barrier. But on the plus side, the second garbage collector does not restrain the amount of memory, and its write barrier is only lightly used. The second collector can also be extended to work with other types of realtime tasks. Ways to make the collectors more cache-friendly have been discussed as well.

Benchmarks show that conservative garbage collectors can be used in programs generating individual samples in realtime without notable loss in performance, at the price of high memory overhead if only one instrument is running. Furthermore, taking snapshots doesn't take much time, and using pointer-containing heaps in the range of 10MB to 100MB probably works on newer computers without setting high audio latency. (The memory bandwidth of an Intel Core i7 system is 25.6GB/s, while the computer running the benchmarks performed at 2.4GB/s.) Considering the 2.8kB of pointer-containing memory used by the MIDI synthesizer, 100MB should cover most needs. Furthermore, the limiting factor is memory bandwidth, which can relatively easily increase in future systems, e.g. by using wider buses.

As a final note, since realtime video signal processing is similar to audio in which a fixed amount of data is produced at regular intervals, the same techniques described in this paper should work for video processing as well.

8. ACKNOWLEDGMENT

Many thanks to David Jeske, Jøran Rudi, Henrik Sundt, Anders Vinjar and Hans Wilmers for comments and suggestions. A special thanks to Cristian Prisacariu for helping to get the paper on a better track in the early stage of writing. Also many thanks to all the anonymous reviewers from the ISMM2009 conference[5] and this ICMC conference.

9. REFERENCES

[1] J. Auerbach, D. F. Bacon, F. Bömers, and P. Cheng, "Real-time Music Synthesis In Java Using The Metronome Garbage Collector," in *Proceedings of the International Computer Music Conference*, 2007.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization." ACM Press, 2003, pp. 285–298.

[3] A. Blackwell and N. Collins, "The programming language as a musical instrument," in *In Proceedings of PPIG05 (Psychology of Programming Interest Group*, 2005, pp. 120–130.

[4] H.-J. Boehm, "A garbage collector for C and C++," http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[5] M. S. Johnstone, A. Paul, and R. Wilson, "Non-compacting memory allocation and real-time garbage collection," Tech. Rep., 1997.

[6] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in *ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 79–86.

[7] K. Matheussen, "Realtime music programming using Snd-Rt," in *Proceedings of the International Computer Music Conference*, 2008, pp. 379–382.

[8] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 2, pp. 61–68, 2002.

[9] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.

[10] J. M. Siskind, Stalin - a STAtic Language ImplementatioN, http://cobweb.ecn.purdue.edu/~qobi/software.html.

[11] G. D. Wakefield, "Vessel: A Platform for Computer Music Composition," Master's thesis, Media Arts & Technology program, University of California Santa Barbara, USA, 2007.

[5] A prior version of this paper was submitted to ISMM2009.