UNIVERSITY OF OSLO
Department of Informatics

Realtime Audio with Garbage Collection

Master Thesis

Kjetil Svalastog Matheussen

November 1, 2010

Abstract

Two non-moving concurrent garbage collectors tailored for realtime audio processing are described. Both collectors work on copies of the heap to avoid cache misses and audio-disruptive synchronizations. Both collectors are targeted at multiprocessor personal computers.

The first garbage collector works in uncooperative environments, and can replace Hans Boehm's conservative garbage collector for C and C++. The collector does not access the virtual memory system. Neither does it use a read barrier or a write barrier.

The second garbage collector replicates the heap using a novel algorithm. The second collector works in similar environments as the first collector, but it requires a write barrier. Two variants of the collector are described, each using a different type of write barrier. Neither of the write barriers should affect audio processing performance.

Acknowledgments

Some of the research for this thesis have been performed at the Norwegian Center for Technology in Music and Art (NOTAM).

Thanks to my adviser Herman Ruge Jervell for inspiration, support, and guidance.

Thanks to my family and the Balstad/Skjetne family for helping with moving and housing during this period.

Thanks to the Fulbright Foundation for sponsoring my stay at the CCRMA center at Stanford University. Some of the courses I took at Stanford University are also part of the Master's degree taken here at the Department of Informatics.

Thanks to Bill Schottstaedt for his work on SND and CLM, Jeffrey Mark Siskind for advices on using his Stalin Scheme compiler, Yann Orlarey for opening my eyes about the memory bus problem, my colleagues at NOTAM (including Anders Vinjar) for helping with the ICMC presentation, and Christian Prisacariu and David Jeske for reading early drafts of the ICMC paper.

Contents

List of Figures

List of Programs

List of Tables

1 Introduction

This chapter introduces garbage collection and realtime audio processing, and the problems that can occur when combining them.

1.1 Garbage Collection

A garbage collector is used to automatically free memory in computer programs.

Garbage collectors can be divided into two types. The first type is the *tracing collector*. A tracing collector finds all objects which can be traced back to the roots[1] and frees everything else [McC60].

The second type is the *reference counting collector*.[2] The reference counting collector uses a counter for each allocated object to keep track of the number of references to an object. When the counter for an object reaches zero, the object can be freed [Col60].

The work in this thesis mostly focuses on tracing collectors. Since programs using reference counting collectors must update the reference counters manually, tracing collectors have potential for being simpler to use, and simpler to make efficient.

[1] The roots are the starting point of a tracing operation. Typical roots are global variables and values in CPU registers.

[2] In earlier papers, for example in [Ste75], garbage collection and reference counting were categorized as two separate types of methods to reclaim memory. But in today's literature, reference counting collectors are categorized as a type of garbage collector, and the term tracing collector has taken over the meaning of what was earlier meant by a garbage collector.

Realtime

The simplest kind of tracing collector traverses all reachable memory in one operation. To prevent references from being overwritten while the collector traverses memory, the program (called the *mutator* [DLM+78]) must be paused during the entire operation. Roger Henriksson accurately characterizes realtime applications by their "need to guarantee short response times to external events and to run continuously without faults for very long periods." [Hen94][3] The simplest kind of tracing collector can not guarantee short response time since the amount of memory to trace can be large. Thus this type of garbage collector does not work well for realtime applications.

One way to avoid long pauses in the execution is to divide tracing into many smaller operations, each operation doing a little bit of work. To avoid losing references, the mutator either uses a *read barrier* to redirect heap accesses or a *write barrier* to track heap changes. This type of garbage collector is known as an *incremental collector* [Wil92, 17].

Another way to avoid long pauses is to run garbage collection in parallel with the mutator [Ste75]. This type of garbage collector is known as a *concurrent collector*. A concurrent collector usually needs a read barrier or a write barrier to synchronize the collector with the mutator.

To increase audio processing performance, the work in this thesis focuses on doing garbage collection in parallel.

1.2 Audio Processing

By tailoring garbage collectors for audio processing, we hope to achieve higher performance and reliability. To find features of audio code, we look at code in SND[4] (version 10.7, released 6th of July 2009).

SND is both a sound editor and a programming environment for sound. 294 *instruments* written for the programming language Scheme are included. An instrument in SND is a function which produces or manipulates sound. Most of the instruments in SND are implemented by William Schottstaedt, but not all of them are invented by him.

[3] Roger Henriksson's exact quote from [Hen94]:

> A real time application is characterized by the need to guarantee short response times to external events and to run continuously without faults for very long periods.

[4] http://ccrma.stanford.edu/software/snd/

SND was chosen as study object because of familiarity, that there is a large number of instruments available, and that instruments are straight forwardly written. Instruments in other music programming systems are often divided into at least two functions. They are divided to avoid allocation and initialization to be performed in the realtime thread. SND, on the other hand, usually allocates, initialize and process sound in the same function.

The instruments in SND are not written to run in realtime, but by having a realtime garbage collection, it is usually simple to translate an SND instrument into doing so.

Looking at instruments in SND, we find that:

1. Based on the first 19 instruments (they seems representative for the rest), the average length of an instrument is 45 lines of code. On average, 10.3 pointers are referenced from inside inner loops. The median is 8 pointers.[5]

2. Both reading and writing to allocated memory is common in those parts of the code where samples are processed (the inner loops).

3. Neither of the 294 instruments writes pointers inside inner loops.[6]

Program 1 shows the source code of an instruments in SND which plays Dual Tone Multiple Frequencies (DTMF) sounds. DTMF sounds are used for dialing telephone numbers. Data used for calculating sound is allocated and initialized between line 7 and line 19. The sound itself is calculated between line 22 and line 25 (this is the inner loop).

Realtime

To better know the advantage of using a garbage collector for realtime audio processing, and to know what it can be used for, we will do a brief study of a realtime instrument.

Program 2 uses the algorithm from program 1 to play DTMF sounds in realtime.[7] Program 2 works by spawning two new coroutines [Con63, DN66] each time a *note on* MIDI message is received. The first coroutine produces sound (line 7), and the second coroutine waits for a *note off* MIDI message (line11).

[5]Name of instrument(number of referenced pointers): pluck(2), vox(21), fofins(4), fm-trumpet(13), pqw-vox(57), stereo-flute(7), fm-bell(8), fm-insect(6), fm-drum(9), gong(9), attract(0), pqw(10), tubebell(7), wurley(8), rhodey(7), hammondoid(5), metal(8), drone(4), canter(11).

[6]The source code of all 294 instruments have been checked.

[7]Making it possible to dial phone numbers using a MIDI keyboard. Program 2 is written for the SND-RT music programming system [Mat10].

Program 1 DTMF Instrument in SND. The Instrument is Written by Bill Schottstaedt.

```
1
2  (definstrument (touch-tone start telephone-number)
3    (let ((touch-tab-1 '(0 697 697 697 770 770 770 852 852 852 941 941 941))
4          (touch-tab-2 '(0 1209 1336 1477 1209 1336 1477 1209 1336 1477 1209 1336 1477)))
5      (do ((i 0 (+ i 1)))
6          ((= i (length telephone-number)))
7        (let* ((k (list-ref telephone-number i))
8               (beg (seconds->samples (+ start (* i .4))))
9               (end (+ beg (seconds->samples .3)))
10              (i (if (number? k)
11                     (if (not (= 0 k))
12                         k
13                         11)
14                     (if (eq? k '*)
15                         10
16                         12)))
17              (frq1 (make-oscil :frequency (list-ref touch-tab-1 i)))
18              (frq2 (make-oscil :frequency (list-ref touch-tab-2 i))))
19          (ws-interrupt?)
20          (run
21           (lambda ()
22             (do ((j beg (+ 1 j)))
23                 ((= j end))
24               (outa j (* 0.1 (+ (oscil frq1) (oscil frq2)))))))))))
25
```

Program 2 Realtime Instrument Playing DTMF Tones.

```
1  (<rt-stalin>
2    (define touch-tab-1 '( 697  697  697  770  770  770  852  852  852  941  941  941))
3    (define touch-tab-2 '(1209 1336 1477 1209 1336 1477 1209 1336 1477 1209 1336 1477))
4    (while #t
5      (wait-midi :command note-on
6        (define number (modulo (midi-note) 12))
7        (define tone (sound
8                      (out (* (midi-vol)
9                              (+ (oscil :freq (list-ref touch-tab-1 number))
10                               (oscil :freq (list-ref touch-tab-2 number)))))))
11       (spawn
12         (wait-midi :command note-off :note (midi-note)
13           (stop tone))))))
```

We compare program 2 with fairly equivalent code implemented in C. Program 3 uses a system called "Realtime sound system" to handle coroutines, realtime memory allocation, MIDI and audio communication.[8] We see a clear difference in program size between program 2 and program 3, although they do exactly the same. Adding a garbage collector to the C version

[8] Although it would be possible to implement "Realtime sound system", it is not a real system, but invented only for this section in order to do a comparison between Scheme and C for audio programming: We have provided "Realtime sound system" with the same functionality as is provided to program 2 by SND-RT.

Program 3 The DTMF MIDI Soft Synth Implemented in C

```
#include <clm.h>
#include <realtime_sound_system.h>

typedef struct tone_t{
  mus_t *oscil1;
  mus_t *oscil2;
  float  volume;
} tone_t;

int touch_tab_1[] = { 697,  697,  697,  770,  770,  770,  852,  852,  852,  941,  941,  941};
int touch_tab_2[] = {1209, 1336, 1477, 1209, 1336, 1477, 1209, 1336, 1477, 1209, 1336, 1477};

void tone_player(float **samples,int num_samples, void* arg){
  tone_t *tone = (tone_t*)arg;
  int i;
  for(i = 0 ; i < num_samples ; i++){
    float sample    = tone->volume * (mus_oscil(tone->oscil1) + mus_oscil(tone->oscil2));
    samples[0][i]   = sample; // Left channel
    samples[1][i]   = sample; // Right channel
  }
}

void player_coroutine(void *arg){
  midi_t *midi_on  = (midi_t*)arg;
  tone_t *tone     = realtime_alloc(sizeof(tone_t));
  int number       = midi_on->note % 12;
  tone->oscil1     = make_oscil(touch_tab_1[number]);
  tone->oscil2     = make_oscil(touch_tab_2[number]);
  tone->volume     = midi_on->vol;
  player_t *player = add_sound(tone_player,tone);
  while(true){
    midi_t *midi_off = wait_midi();
    if(midi_off->type==COMMAND_NOTE_OFF && midi_off->note==midi_on->note){
      remove_sound(player);
      free_oscil(tone->oscil1);
      free_oscil(tone->oscil2);
      realtime_free(tone);
      realtime_free(midi_on);
      realtime_free(midi_off);
      return;
    }else
      realtime_free(midi_off);
  }
}

void realtime_process(){
  while(true){
    midi_t *midi = wait_midi();
    if(midi->type==COMMAND_NOTE_ON)
      spawn_coroutine(player_coroutine,midi);
    else
      realtime_free(midi);
  }
}

int main(){
  start_realtime_sound_system(realtime_process);
  return 0;
}
```

of the MIDI soft synth would only make the program 9 lines shorter, while the Scheme version could not have been written without a garbage collector. (Data are stored implicitly in closures in the Scheme version.[9])

In addition, while a Scheme program can implement coroutines by using continuations, a C program is forced to implement coroutines by changing system execution context, for instance by switching stack and calling *setjmp* and *longjmp*. Coroutines using continuations can be made faster in Scheme (and other functional languages) since switching execution can be as simple as doing a jump.

1.3 Predictability

The goal of this thesis is to create garbage collectors that minimize the chance of interrupts in the stream of data sent to the soundcard. To do that, it is useful to know how a soundcard works. Inside a soundcard there is a small hardware buffer which must be filled with new data at regular intervals. When the soundcard is ready to receive new data, an interrupt triggers audio processing code to start working.[10] The amount of data required to fill the soundcard buffer is called a *block*, and the *length* of a block is normally between 32 and 4096 samples per sound channel (known as a *frame*). The samplerate is usually 44100Hz or 48000Hz.

For a soundcard running with a samplerate of 48000Hz, and a block length of 32 frames, the *duration* of a block will be 0.67ms. Having a higher samplerate than 48000Hz, or a lower block length than 32 frames, is uncommon.

In case audio code is unable to fill the hardware buffer fast enough, one of two things normally happens: 1. A block of empty data is played, or: 2. The previous block is played one more time. The sound we hear when this happens is called a *glitch*.

To avoid glitches, audio code must be coded efficiently, but equally important is that the audio code behaves predictably. For example, it is expected that a program will fail if it runs too many oscillators at once, but we don't want to expect a program to maybe fail if we repeatedly start and stop 1000 oscillator. The program must behave consistently. If the program succeeds

[9] *while*, *spawn*, *sound* and *wait-midi* are macros which puts their bodies into lambda blocks.

[10] It should also be noted that audio code are usually not communicating directly with the soundcard. These days, audio code commonly communicates with a sound system such as JACK for Linux [LFO05], Core Audio for Mac OS X, or DirectSound or ASIO for Windows. These systems provide higher level interfaces to the soundcard, but the principle of generating blocks of samples at regular intervals is similar.

the first time, it must also succeed all subsequent times. And similarly, if the program fails the first time, it must also fail all subsequent times.

Two Criteria for Predictability

One criteria for a garbage collector to perform predictably is that there must be an upper bound on the time used to allocate one block of memory, regardless of the amount, or type, of previously allocated memory.

Another criteria is that there must be an upper bound on the accumulated time spent by the garbage collector within each block, regardless of the state of the heap.

Lower Bound and Upper Bound

The two criteria above defined upper bounds both on allocating time and garbage collecting time. But for interactive usage, we also need a lower bound which is regularly equal to the upper bound.

Here is an example: Lets say the upper bound for the time spent collecting garbage is 0.09ms per allocation. During rehearsal for a concert, the time spent collecting garbage was seldom as high as 0.09ms, and no glitches were heard. At the concert, almost the same code was run, but this time occasional clusters of 0.09ms was spent collecting garbage, and the audience heard glitches. A garbage collector specified to support hard realtime by guaranteeing an upper bound did not perform adequately. In order to use this garbage collector correctly, it would have been necessary for the user to investigate when and how often memory was allocated. But even if it was possible, a musician who wants to focus on performing music should not have to do this. Especially if the musician used ready-made software and had no technical training.

The example above does not tell us that lower bound must always be equal to upper bound, but rather that the user needs to hear a glitch during rehearsal in order to know when the system is pushed too far. But it would be impractical if a glitch only appeared at the end of the rehearsal, and maybe the rehearsal would have to be repeated. So in this thesis, 1 second is the longest amount of time until the lower bound must be equal to the upper bound. In other words, worst case must happen at least once a second.

For the example above, 1 second may sound lower than necessary. But to better know the limitations of the software, it is an advantage to have a

fast feedback.[11]

1.4 Efficiency

Besides keeping deadlines and behaving predictably, a garbage collector should also avoid making audio code run slower.

To maintain efficiency of audio processing algorithms, we need to know where time is spent. When the audio code fills a block of audio, it is common to go through several *for* loops which produces or manipulates samples. Each of these loops iterates between 32 and 4096 frames. We call these loops Digital Signal Processing loops (*DSP loops*). One example of a DSP loop was present in the function *tone_player* in program 3. Another example was present between line 7 and 11 in program 2.

Since 44100 or 48000 number of samples are usually processed per second, other operations, such as handling user events or selecting which DSP loop to run, are likely to happen far less often than the code running inside the DSP loops.

Read and Write Barriers

It is common for realtime garbage collectors to use a write barrier to track heap changes, or a read barrier to redirect heap accesses. These barriers are often used so that memory can be reclaimed incrementally or concurrently. A summary of barrier techniques for incremental garbage collection can be found in [Pir98].

The main performance concern of using a read or a write barrier for audio code, is if barrier code is added inside DSP loops. Not only will the extra instructions required by a read or write barrier be executed for each sample, but barriers can also destroy vectorizations [SOLF03].

Another problem is unpredictable cache misses. Read barriers and write barriers increase the number of instructions performed by the mutator, and by putting less pressure on the CPU cache, the CPU cache may behave more predictably.

Write Barriers

In section 1.2 we saw that no pointers were written inside DSP loops. Therefore, a write barrier should not decrease performance of DSP loops if the

[11] The value of 1 second is used throughout this thesis because it seems like an appropriate duration. But in an implementation, it would be natural if the value could be adjusted by the user.

write barrier is used only for writing pointers. This means that the largest disadvantage of using a write barrier for audio programming is that it becomes hard to use existing languages if they lack support for write barriers.

(At this point, we also introduce three new terms: *L-value*, *R-value*, and *Old R-Value*. These are the values that a write barrier can operate on. The C program listed in program 4 prints these three values to the terminal for the assignment $a = 5$.)

Program 4 Printing *L-value*, *Old R-Value*, and *R-Value*

```
#include <stdio.h>

int a = 0;

int main(){
  printf("  L-Value:     %p\n", &a);
  printf("  Old R-value: %d\n",  a);

  a = 5;

  printf("  R-Value:     %d\n",  a);
  return 0;
}

/*
  L-Value:     0x80496a8
  Old R-value: 0
  R-Value:     5
*/
```

Optimizing a Read Barrier

One way to prevent read barriers from interfering with DSP loops, is to use temporary local variables to prevent pointers from being accessed inside a DSP loop. Program 5 shows an example of such code transformation.

However, this optimization contributes extra complexity, and it can be hard to analyze code to locate DSP loops correctly. And for arrays of pointers, pointers to pointers, and so forth, the optimization becomes increasingly

difficult to apply.[12]

Program 5 A Sine Oscillator DSP loop

```
// Original version:
void tone_player(float **samples,int num_samples, tone_t *tone){
  int i;
  for(i = 0 ; i < num_samples ; i++){
    float sample   = tone->volume * sin(tone->phase);
    samples[0][i]  = sample;
    samples[1][i]  = sample;
    tone->phase   += tone->phase_inc;
  }
}

// Same function, but transformed to minimize read barrier impact:
void tone_player(float **samples,int num_samples, tone_t *tone){
  int i;
  double tone_volume    = tone->volume;
  double tone_phase     = tone->phase;
  double tone_phase_inc = tone->phase_inc;
  float *samples0       = samples[0];
  float *samples1       = samples[1];
  for(i = 0 ; i < num_samples ; i++){
    float sample = tone_volume * sin(tone_phase);
    samples0[i]  = sample;
    samples1[i]  = sample;
    tone_phase  += tone_phase_inc;
  }
  tone->phase = tone_phase;
}
```

1.5 Conservative

A conservative garbage collector considers all values in the heap and the roots as potentially being a pointer [BW88]. The advantage is that the mutator doesn't have to specify exactly where pointers are placed. This means that

[12] Even for program 5, which was manually transformed to lower read barrier impact, it was impossible to remove the sample arrays *samples*0 and *samples*1 from the DSP loop.

if the collector finds a value that could be a pointer, it takes no chance, and behaves as if it is a pointer. Hence the word conservative.

A number of efficient language implementations, such as *Stalin Scheme*, *Bigloo Scheme*, *D*, and *Mono*, uses the conservative Boehm-Demers-Weiser garbage collector for C and C++ [Boe].[13] (BDW-GC)

Since DSP operations require much processing power, it is important to be able to use efficient languages. And since conservative collectors are simple, it is generally much easier to replace a conservative garbage collector in a language implementation, than replacing other types of collectors.

There are two issues with conservative collectors running in realtime:

1. Fragmentation in the memory heap can cause programs to run out of memory prematurely. Since the garbage collector can not know whether a value is a pointer or not, it can not overwrite pointers either, which would have been necessary in order to move objects in memory, and avoid fragmentation.

2. Memory leakage caused by; 1. Values in memory misinterpreted as pointers (*false pointers*), and; 2. Pointers which are not used anymore, but visible to the collector.

False pointers should not be a problem on machines with 32 bit or higher address space, but pointers which are not used anymore can be. One example is a program that removes the first element from a singly-linked list, but leaves the *next* pointer in the element uncleared. If the list becomes garbage while the removed element is still alive, the rest of the list is still traceable because of that *next* pointer. Rafkind et al. have reported that the PLT Scheme implementation sometimes ran out of memory after a few hours, before they switched to precise garbage collection [RWRF09].[14] However, a large program such as PLT Scheme is likely to use magnitudes more memory

[13] These language implementations have been among the fastest at "The Computer Language Benchmarks Game". (`http://shootout.alioth.debian.org/`.) However, Stalin Scheme, Bigloo Scheme and D are not listed anymore.

A few simple benchmarks for Stalin have been performed in [Gra06]. In these benchmarks, Stalin performed approximately similar to C programs compiled with GCC 4.02. Stalin also performed significantly faster than SML (MLton 20041109) and Haskell (ghc 6.4.1)

[14] "Precise" is here the opposite of "conservative". Rafkind et al. have created a garbage collector that transform the source code of a C program so that it keeps directly track of when memory becomes unavailable. The collector seems to be an automatic reference counting collector. This type of collector can actually work in uncooperative environments (that's what it was made for), but the resulting code after transformation looks verbose and might not be suitable for efficient audio processing.

than the kind of instruments we looked at earlier in this chapter, and therefore more likely to experience problems. Rafkind et al. writes further that the conservative collector "seemed to work well enough" while DrScheme (PLT's programming environment) was a smaller program. Apparently, threading was one of the main causes for the memory leaks since thread objects link to each other and stores stacks and registers information. (Memory usage and leakages in conservative garbage collectors are further discussions in [Boe02]).

Fragmentation has been analyzed in several types of programs by Johnstone and Wilson. Johnstone and Wilson concluded [JW99] by saying that "the fragmentation problem is really a problem of poor allocator implementations, and that for these programs well-known policies suffer from almost no true fragmentation." Bacon, Cheng, and Rajan believe the measurements of Johnstone and Wilson "do not apply to long-running systems like continuous-loop embedded devices, PDAs, or web servers." [BCR03]. For our usage, interactive audio, where the user has rehearsed and learned the behavor of a system, surprises can be avoided by giving a warning when a program risks running out of memory. This is possible if the memory allocator has predictable or reasonably bounded fragmentation.

1.6 Snapshot-at-the-Beginning

This section presents a garbage collector that; 1. Has an upper bound on execution time, and; 2. Can be made conservative.

If we use hardware to track changes in memory, we can create a realtime collector without having to insert read or write barriers into the source code.

One example is a concurrent collector which uses the virtual memory system to function as a write barrier. This type of collector uses a technique called *copy-on-write* to make a *snapshot* of the heap[15] [Wil92, 20]. A snapshot is in this case a virtual consistent state of the heap. Instead of physically copying memory to create a snapshot, copy-on-write creates a new set of virtual memory pages which maps back to the memory. Only when the mutator writes to new parts of the heap, memory will be copied physically.

[15] In Unix and Unix compatible operating systems [Uni97], a write-barrier based on using copy-on-write can be implemented using $fork()$. $fork()$ requires all memory to be copied to the new process, and it usually uses copy-on-write to do so. Using copy-on-write for $fork()$ both reduces the time to fork, and the needed amount of physical memory.

When the garbage collector finds unreferenced memory in the child process, pointers to free memory blocks can be sent to the parent via an IPC mechanism. A garbage collector working like this is described by Rodriguez-Rivera and Vincent F. Russo in [RRR97].

Another collector using copy-on-write is [AEL88], which uses copy-on-write in the implementation of a concurrent copying collector.

In other words, copy-on-write works as a write barrier that writes the *Old R-Value* into the snapshot, using the *L-Value* to find write position.

When we have a snapshot like this, the collector can work freely in parallel without disrupting audio processing.

Although the code becomes simpler with a write barrier implemented in hardware, the amount of time spent by the virtual memory system will vary. In some blocks, nothing could be copied, while in other blocks, the entire heap could be copied. And whenever the virtual memory has to copy a page, the mutator has to wait until the copying of the page is complete. Thus, this kind of collector can cause glitches since there is no lower-bound, only an upper-bound.

Additionally, unforeseen variations and slow-downs in execution can happen depending on how the operating system handles exceptions raised by the memory management unit.

1.7 Chapter Summary

This chapter has described features of realtime audio processing and problems when making a garbage collector for realtime audio processing. The chapter ended by describing a type of garbage collector that could almost be made to work for realtime audio processing both in terms of performance, ease of use, and reliability.

1.8 Definition of Terms

Atomic memory
Memory not containing pointers to other allocated memory blocks.

Atomic operation
An operation which completes before another CPU can alter the result in the meantime.

The most common atomic CPU operation is assigning values to memory locations. For instance, the following code works atomically on most types of CPUs: "a=5", where the number of bits in a is equal to the number of bits used by the CPU. Right after this operation is finished, a will always have the value 5, and not a combination of 5 and some other value which might have been written to a at the same time.

Another common atomic operation is the Compare and Swap instruction (CAS). *CAS(L-Value,a,b)* will write b to the *L-Value* if a is equal to the *Old R-value*. CAS is commonly used to atomically increment or decrement an integer by running a CAS instruction in loop until a write was successfully performed, but it can also be used to shade the color of a memory object in a parallel garbage collector.

Collector
Part of a program that collects garbage (i.e. the garbage collector).

Conservative garbage collector
A collector that does not know exactly where pointers are placed in memory. Therefore it must treat all values in the heap and the roots as potentially being pointers.

Copying garbage collector
A collector that copies memory to new positions. Usually in order to avoid fragmentation.

Block
Amount of sound data processed between soundcard interrupts. Common size is between 32 and 4096 frames.

Frame
Smallest time unit when processing samples:

- A block usually consists of between 32 and 4096 frames.
- A soundcard commonly plays 44100 or 4800 frames per second.

- A soundcard plays n number of samples per frame, where n is the number of sound channels.

Mutator
Part of a program that writes to the heap, uses and allocates memory, and runs the normal part of a program.

Non-moving garbage collector
A garbage collector that will not move memory to new positions. The opposite of a copying garbage collector.

Roots
Initial data a tracing collector starts working from. Typical roots are the program stack, CPU registers, and global variables.

Glitch
What is heard when sound data are not produced fast enough.

L-value
In the assignment $a = b$, the address of a is the L-Value.

Old R-value
In the assignment $a = b$, the value of a before the assignment is the Old R-Value.

R-value
In the assignment $a = b$, b is the R-Value.

Snapshot
A snapshot is a copy of the heap, taken at one exact point in time.

Samplerate
Number of frames processed per second by the soundcard. Common values are 44100 and 48000.

Uncooperative environment
The term used by Hans-Juergen Boehm and Mark Weiser in [BW88] to denote the environment a common C program runs in. See also *conservative garbage collector*.

1.9 Document Overview

Chapter 1 introduced garbage collectors and realtime audio processing, including problems that can occur when combining them.

Chapter 2 gives a brief history of garbage collection.

Chapter 3 introduces software which are used by programs in this thesis.

Chapter 4 describes semantics and syntax of programming code in this thesis.

Chapter 5 describes the first garbage collector.

Chapter 6 shows a simple implementation of the first garbage collector.

Chapter 7 introduces a minimal memory allocator and analyzes fragmentation.

Chapter 8 extends the implementation of the first collector to support more than one heap, and improves it so that several collectors can run simultaneously without risking unpredictable performance.

Chapter 9 extends the first collector to run faster and more predictably.

Chapter 10 runs several tests to find snapshot performance under different conditions, and uses data from these tests to improve the first garbage collector.

Chapter 11 compares the first garbage collector with a conservative mark-and-sweep collector.

Chapter 12 describes the second garbage collector.

Chapter 13 concludes the thesis and lists achievements and future work.

Appendix A contains source codes. The benchmark program and the programs to test snapshot performance are included here.

Appendix B lists all benchmark data and shows the remaining figures generated by the snapshot performance tests.

In addition, two papers are applied at the end the document. The first paper is from the International Computer Music Conference 2009, which contains some of the same material as in this thesis, but less clearly. The second paper is from the Linux Audio Conference 2010, and shows practical

usage for the first garbage collector when used in audio programs. The second paper also documents better how program 2 works, and how the MIDI synthesizer in appendix A.1 works.

2 Brief History of Garbage Collection

This chapter gives a brief history of garbage collectors from 1960 to 2010. An emphasis is given to realtime collectors.

The summary is far from complete. Most notable is the omission of reference counting collectors created after 1960, such as [Bak94, BR01, BM03, LP06].

1960: The Mark-and-Sweep Collector

Garbage collection was first described by John McCarthy in [McC60].

The described method, later commonly known as *mark-and-sweep*, had two phases. In the first phase, the collector went recursively through the reachable memory marking all elements by setting a sign. The second phase iterated over all allocated elements and inserted non-signed elements into a list of free elements.

1960: Reference Counting

McCartney's method had one problem. Garbage collection could pause the execution of a program for very long periods.

George E. Collins' solution to this was to include a counter with each memory object. This counter was increased by one when a reference was made to it. And similarly, the counter was decreased by one if a reference was deleted. This made it possible to insert objects into the free-list at the exact moment they became garbage [Col60].

1970: Semispace Collector

Robert R. Fenichel and Jerome C. Yochelson described in 1969 a garbage collector which used two "semispaces" [FY69]. The heap was divided into two areas of memory, known as *from-space* and *to-space*. Only *from-space* was used by the mutator.

When *from-space* became full, the garbage collector moved all live memory from *from-space* into *to-space* and updated pointers to the new positions. When finished, only garbage would be left in *from-space*. The last step was to switch the position of the *from-space* and *to-space* (known as *flipping*).

C.J. Cheney described an improved, but still simple, non-recursive semispace collector in 1970 [Che70], and Marvin Minsky described a similar collector already in 1963 [Min63]. However, Minsky used an external disk as intermediate storage [Wil92].

One advantage of this method compared to mark-and-sweep is that it's simple to allocate objects of different sizes. If *from-space* and *to-space* are continuous blocks of memory, allocating a new block is just increasing a pointer, and returning the old position. In non-moving collectors, such as a mark-and-sweep, it is necessary to search around in memory in order to find a fitting area of memory which is large enough in order to achieve lowest possible fragmentation. Unpredictable fragmentation is also avoided in a semispace collector since the collector always uses twice as much memory.

1975-1978: Concurrent Collectors with Short Pauses

Between 1975 and 1978, a series of concurrent collectors were described.

Guy L. Steele Jr. published the first paper in 1975 named "Multiprocessing Compactifying Garbage Collection" [Ste75]. A write barrier was used to store the *Old R-Value* into a "gc stack". Freshly allocated objects were also pushed into this gc stack. The collector popped and pushed objects off the gc stack while marking objects. When the gc stack became empty, memory was compacted, and unreferenced memory freed. Both semaphores and spinlocks (called *munch*) were used to synchronize the collector and the mutator. A read barrier was used to redirect newly compacted memory by forwarding to another memory position if the pointer was a forwarding pointer (the collector used an extended type of pointer to distinguish directly pointing pointers from forward pointing pointers).

Another influential paper was published in 1978 by Dijkstra, Lamport, Leslie, Martin, Scholten, and Steffens. This was the first paper to use the terms *mutator* and *collector*. The paper was also the first to give objects different "colors" depending on how far they were in the collection process.

The technique, later known as *tricolor marking* [Wil92, 18], divides objects into white, gray and black. A white object is an object that has not yet been traced. A gray object is an object which has been traced, but not its descendants. A black object is an object where both the object itself and its descendants have been traced. The tricolor marking is used for bookkeeping so that garbage collection can be divided into smaller operations. When there are no more gray objects, the remaining white objects are garbage. The write barrier in Dijkstra et al.'s collector *grayed* (i.e. made sure an object became gray in case it was white) the *R-Value* to avoid losing references in case all objects pointing to the *L-Value* had already been traced.

A short overview of concurrent garbage collectors from this period is found in [Bak78].

1978: Baker's Incremental Realtime Copying Collector

Henry G. Baker described in 1978 a realtime garbage collector running on uniprocessor computers [Bak78]. This collector was a semispace collector, but instead of moving all objects and collecting all garbage in one operation, the job was divided into many smaller operations.

A read barrier moved objects into *to-space*, and left a forwarding pointer in the old position. When memory was allocated (using "CONS"), a certain amount of memory was traced (and copied) using *to-space* and roots as starting point. Memory was allocated from the top of *to-space*, while live objects were copied into the bottom of *to-space*. A pointer was used to track both positions. When the two pointers met, *from-space* and *to-space* were flipped. By making sure that "CONS" traced enough memory, all live memory would be moved into *to-space* before flipping.

1983: Generational Garbage Collector

Henry Lieberman and Carl Hewitt made a new garbage collector in 1983 based on the hypothesis that newly allocated memory became garbage faster than older memory [LH83].

Although worst-case performance for a generational collector is equal to a mark-and-sweep collector, long pauses generally happens so seldom that it can be used for interactive programs.

1984: A Cheap Read Barrier for Baker's Collector

Rodney A. Brooks modified Baker's incremental garbage collector from 1978 so that more work was done by the write barrier than the read barrier [Bro84]. In order for the mutator to get hold of objects in *to-space*, the collector used

a forwarding pointer as read barrier. Since all pointers in the system were forwarding pointers, the read barrier had no branching and was very efficient.

1987: Snapshot-at-the-Beginning Collectors

Snapshot-at-the-beginning collectors creates a virtual consistent state of the heap in order to collect garbage concurrently or incrementally. A write barrier often stores the *Old R-Value*, or mark the contents of the *Old R-Value*.

Taichi Yuasa published a paper in 1990, where the *Old R-Value* was pushed into a stack so that it could be marked later [Wil92, 20].

Another type of snapshot-at-the-beginning collector uses the virtual memory system to automatically copy parts of the heap which are modified by the mutator, (apparently) first described by Abrahamson and Patel in 1987 [Wil92, 20]. This type of collector is also described in section 1.6.

1988: Conservative Collector

Hans-Juergen Boehm and Mark Weiser described in 1988 a mark-and-sweep collector for C and C++ [BW88]. This collector could be used in already existing programs just by redirecting *malloc* and *calloc* to their respective alternatives, and using a dummy macro or an empty function for *free*. In order to make this work, it was necessary for *mark* to scan memory conservatively, meaning that if a value in memory looked like a pointer, it also had to be treated as a pointer.

1992: The Treadmill Collector

Henry G. Baker described in 1992 a non-moving version of his realtime collector from 1978. Instead of moving the memory itself, this collector only moved pointers to memory [Bak92].

1992: Realtime Replicating Collector

Scott Nettles and James O'Toole described in 1993 a replicating garbage collector capable of realtime operation [NO93]. A replicating garbage collector is an incremental copying collector which does not modify *from-space*. A write barrier ensures that changes in *from-space* are also applied to their copies in *to-space*. Before flipping, pointers in the roots are updated to point to the new positions in *to-space*. Since *from-space* is not modified, it is tempting to implement a concurrent replicating collector, which Nettles and O'Toole also did [ON94].

1993: The Wilson-Johnstone collector

Paul R. Wilson and Mark S. Johnstone described in 1993 another non-moving incremental realtime collector [WJ93]. This collector had no read barrier, and it was not concurrent. Tricolor marking was used to divide garbage collection into many smaller operations.

The collector works like this: When a new collection cycle starts, all objects are white, and the root set is marked gray. A bit of memory is traced each time the mutator allocates memory. Similar to Dijkstra et. al's collector, a write barrier is used to gray the *R-Value* to ensure it won't disappear, in case all references to the *L-Value* had already been traced. When there are no more gray objects, the remaining white objects can be recycled. Since there is no read barrier, and the amount of time spent collecting garbage depends on the amount of requested memory (allocating 5000 bytes takes longer time than allocating 500 bytes), this collector is likely to perform quite predictably and with short pause times.

2005: The Metronome Collector

David F. Bacon, Perry Cheng, and V.T. Rajan described in 2003 a real-time incremental garbage collector which used a scheduler to ensure that the garbage collector only spends a certain amount of time within specified intervals [BCR03]. To avoid fragmentation, memory were sometimes moved ("mostly non-copying"). A simple forward-pointer read barrier (similar to Brooks' read barrier from 1984), was used in order to support moving memory.

2001-2010: Recent Concurrent Realtime Collectors

Richard L. Hudson and J. Eliot B. Moss described in 2001 a copying collector which minimized pause times by avoiding threads to synchronize when memory is moved to new positions [HM01].

Filip Pizlo, Daniel Frampton, Erez Petrank and Bjarne Steensgaard described in 2007 a realtime mark-and-sweep collector which compacted memory without requiring any locks (the *STOPLESS* collector) [PFPS07]. Atomic operations were used instead of locks. The year after, Pizlo, Petrank and Steensgaard presented two variants of the same collectors, each having different characteristics [PPS08].

Fridtjof Siebert described in 2010 a realtime collector which was both parallel and concurrent, meaning that several parallel collector threads ran concurrently with the mutator [Sie10]. A very simple write barrier was used to shade objects, and atomic instructions were used to protect objects to

avoid coarse-grained locks. Benchmarks from [Sie10] seemed to indicate that the collector scaled very well with the number of CPUs.

3 Software

3.1 Stalin Scheme

Stalin Scheme [Sis] is an R4RS Scheme compiler [CR91] written by Jeffrey Mark Siskind.

The Stalin compiler applies whole-program optimizations and other optimizations to produce efficient code. According to the documentation, the aim of the compiler is to "generate efficient executable images either for application delivery or for production research runs."

All the realtime Scheme examples in this thesis, including the benchmark program in appendix A.1, are compiled with the Stalin Scheme compiler.

3.2 Rollendurchmesserzeitsammler

The implementation of the first garbage collector is called Rollendurchmesserzeitsammler.

The source code of Rollendurchmesserzeitsammler is available from this world wide web address:

`http://users.notam02.no/~kjetism/rollendurchmesserzeitsammler/`

Rollendurchmesserzeitsammler is made to replace the BDW-GC collector [Boe] in existing language implementations such as Stalin Scheme.

3.3 TLSF

The "Two-Level Segregate Fit" allocator (TLSF) is a memory allocator made by Miguel Masmano, Ismael Ripoll & Alfons Crespo [MRCR04]. TLSF has

constant low allocation time, and it seems to have low fragmentation. Rollendurchmesserzeitsammler can be configured to use TLSF for allocating memory.

Allocation in TLSF works by looking up in a two-dimensional array of cells to find a quite fitted memory block to allocate from. Each cell holds a list of memory blocks. Since two levels of bitmaps are updated to denote which of the cells that have memory available (the second level more fine-grained than the first), finding the most fitting list is only a matter of scanning two bitmaps, an operation which can be performed in $O(1)$. If the found memory block is larger than the requested size (at least over a certain limit), the remaining memory is inserted back to the heap. When a memory block is freed in TLSF, it is merged with memory blocks placed before and after in memory (if possible), before the block is inserted back to the heap.

TLSF achieves low fragmentation since the large number of cells makes it possible to find a quite fitting memory block to allocate from.

3.4 SND and SND-RT

SND is a sound editor and a non-realtime programming environment for sound. SND has CLM built in, which is a sound synthesis package [Sch94]. Both SND and CLM are written by Bill Schottstaedt.

SND-RT is a realtime extension for SND. SND-RT is an experimental audio programming system which uses Stalin for producing realtime audio code and Rollendurchmesserzeitsammler for collecting garbage. The benchmarks programs in this thesis runs inside SND-RT. Similar to SND, SND-RT can also use CLM for sound synthesis.

4 Syntax and Semantics

All code presented in this thesis (except for the pseudocode in chapter 5 and chapter 12) are semantically equivalent to C. (At least most of the time.) The syntax is inspired by Python.

Albeit this chapter does not contain formal syntax and semantics, the code in later chapters should be simple enough so that no semantic misunderstandings should appear.

The main syntactic change from C is that indentation and the colon sign are used instead of { and }, and that all semicolons have been removed. Similarly, some unnecessary parenthesis required by *while*, *if*, and *struct* have been removed as well.

Additional differences from real code:

- Pseudocode appear as plain text without extra syntax. An example: *collect garbage*
- Error checking has been left out.
- Smaller optimizations are left out for clarity. (Such as modifying an existing list instead of building up a new one from the ground.)

Types

Although C is an explicitly typed language, almost all type declarations have been removed. All types should be obvious from their names and usage alone. Hopefully, the code becomes simpler to read when redundant information has been removed.

Accessing Attributes

Dot ('.') is used as infix operator to access attributes: `heap_element.next`. We also ignore that there is a difference in C between `heap_element.next` and `heap_element->next`. There should be no misunderstandings on this matter when reading the code.

Syntax for Single Linked Lists

Single linked lists is the most used data structure.

- The function *put* inserts an element into the start of a single linked list. In C, *put* could be defined like this:

  ```
  #define put(a,b) do{a->next=b;b=a;}while(0)
  ```

- *pop* pops out the first element of a list and returns it. Using the GCC C compiler, *pop* could be defined like this:

  ```
  #define pop(b) ({typeof(b) _temp=b;b=b->next;_temp;})
  ```

- $foreach$ iterates over all elements in a linked list. In a C implementation with even more extended macro capabilities than GCC, $foreach$ could be defined like this:

  ```
  #define foreach(a `in` b ...) \\
    for(a=b;a!=NULL;a=a->next){...}
  ```

- {} is the empty list. In C, the value for {} would be $NULL$.

wait and *signal*

The functions *wait* and *signal* are extensively used in the code. These are operations on semaphores, where the semaphore value is increased when calling *signal* and decreased when calling *wait*. When *wait* is called with a value of 0, *wait* will block until someone else calls *signal* on the semaphore.[1]

is_waiting

is_waiting is used to check if a thread is currently waiting for the semaphore. Since the semaphore can be incremented right before someone starts waiting, it is not enough to check the semaphore value. In the code presented in this thesis, *is_waiting* is supported by extending semaphores like this:

[1] At least as long as the semaphore value is not negative.

```
struct mysemaphore:
  semaphore
  boolean is_waiting

signal(mysemaphore):
  mysemaphore.is_waiting=false
  sys_sem_inc(mysemaphore.semaphore)

wait(mysemaphore):
  mysemaphore.is_waiting=true
  sys_sem_dec(mysemaphore.semaphore)
  mysemaphore.is_waiting=false

is_waiting(mysemaphore):
  return mysemaphore.is_waiting
```

is_ready

is_ready checks whether the semaphore value is higher than 0 (i.e. not waiting), and can be implemented like this:

```
is_ready(mysemaphore):
  return sys_sem_val(mysemaphore.semaphore) > 0
```

5 The First Garbage Collector

This chapter describes the first garbage collector. Based on the arguments in chapter 1, and the problems which were described there, we set up the following list of wishes for the collector:

- Should not use read barriers.
- Write barriers must only be used for writing pointers.
- Should work in uncooperative environments [BW88] in order to easily support efficient languages. This implies:
  - Memory can't be moved.
  - Read or write barriers can't be used.
- Predictable execution: There must be an upper bound, and lower bound must regularly be equal to the upper bound.
- A concurrent collector. Infiltrating audio processing with garbage collection can cool down the CPU cache.

5.1 Creating a snapshot

In section 1.6, we described a concurrent garbage collector that could work in uncooperative environments. The collector also had an upper bound on the amount of time for the mutator to pause. The main problem was that it had no lower bound and could block audio processing for a variable amount of time between each audio block.

We propose a simple adjustment to the collector described in section 1.6. Instead of using copy-on-write to make a snapshot, we copy the heap manually. Copying the heap manually is both simple and has $O(1)$ performance. And for many cases, the amount of memory to copy should not be a problem:

1. Snapshots are not necessary for pointerless memory (*atomic memory*). Example of such memory are sample buffers and FFT data. This is were most memory is spent.

2. Even larger audio programs don't require much pointer-containing memory. In section 1.2, we saw that most of the instruments used less than 9 objects inside the DSP loops. In the DTMF synthesizer implemented in *program 2*, about 7 objects were required per playing tone. If we play 16 tones simultaneously, the number of allocated objects would be 112, and if we multiply that number with the 8 objects used in an average SND instrument, we end up with 1008 pointers. If each pointer takes up 4 bytes, the final amount of memory is 3584 bytes. This would be the memory usage of a small polyphonic MIDI synthesizer.

   Although this is an ideal calculation, it still indicates that the required time to copy the heap for audio programs could be low. Even 3584 bytes multiplied by one hundred takes an almost insignificant amount of time to copy on current personal computers compared to the duration of a common audio block (figure B.1 at page 126 shows snapshot performance).

Another thing to keep in mind is that CPU usage is not what matters in realtime software. As long as taking snapshots doesn't increase the chance of glitches, or reduces interactivity for the user, performance is acceptable. And since multiprocessor computers seldom seems to work to its fullest potential when processing audio, there is a higher chance for CPU time to be available for taking snapshots.

5.2 The Basic Technique

We try to make a predictable garbage collector by taking a snapshot of the heap (by physically copy all memory in the heap in one operation), and run the collector in a parallel thread.

When we find garbage on a snapshot in a parallel thread, the only critical section (where the mutator and the collector are not allowed to run simultaneously), is when we make the snapshot. Snapshots can be created between processing audio blocks.

To avoid worst case performance to increase during execution, the size of the heap can not increase. When the collector starts up, we allocate a fixed size heap small enough to be fully copied within this time frame:

$$m - s \tag{5.1}$$

where m is the duration of one audio block, and s is the time the program uses to process all samples in that block. This heap is used for storing pointer-containing memory.

After taking the snapshot, an independently running lower-priority thread is signaled. The lower-priority thread finds garbage by running a mark-and-sweep on the snapshot.

Program 6 Basic Version of the First Collector

```
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   endif
```

Program 6 shows what has been described so far. Some comments on program 6:

- "*audio function()*" on line 6 is called at regular intervals to process one block of audio.
- The check for "*if there might be garbage*" on line 10 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 CPU usage.
- The *mark-and-sweep* thread on line 1 (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 a high enough priority to prevent GUI updates and other non-realtime operations from delaying memory from being reclaimed.

5.3 Issues

1. By either increasing the heap size, reducing the block size, or increasing the samplerate, the $\frac{time\ creating\ snapshot}{block\ duration}$ ratio is increased, and the time available for audio processing is reduced. (Audio processing can not run simultaneously with taking snapshot.)

2. If a program runs out of memory, and we are forced to increase the size of the heap, the time taking snapshot will increase as well after restarting the program.

Although the situations raised by these issues are unusual, they are not unpredictable.

5.4 Optimizations

The following optimizations can be applied:

- **Running in parallel**
  If instruments or applications depend on each other, for example if one instrument produces audio which is used by a subsequent reverberation instrument, performance can increase if the subsequent reverberation instrument can run simultaneously with taking snapshot. This extension is implemented in program 7.

- **Partial snapshots**
  As discussed in chapter 1, it is only necessary to take snapshot of the complete heap (*full snapshot*) about once a second, or thereabout. In between, only the used part(s) of the heap can be copied instead (*partial snapshot*). Since the soundcard buffer is normally refilled somewhere between 50-1500 times a second, this will eliminate most of the wasted CPU. (At least as long as we don't count taking partial snapshot as wasted CPU.) The user will still notice when the garbage collector spends too much time, but now only once per second, which should be good enough.

- **Avoid taking two snapshots in a row**
  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 is now:

$$2 * (m - s) \tag{5.2}$$

  where m is the duration of one audio block, and s is the duration of processing the samples in that block.

- **Avoid taking snapshot if the audio process takes a long time**
  In case producing audio for any reason takes longer time than usual, which for instance can happen if the CPU cache is cold, sound generators are allocated, or sample buffers are initialized, worst-case can be lowered by delaying a new snapshot until the next block.

Program 7 Basic Version of the First Collector, with 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
    endif
    signal audio function semaphore

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

5.5 Synchronizing Garbage Collectors

If more than one program runs this garbage collector simultaneously, more than one snapshot could also be taken simultaneously. Since personal computers only have one memory bus between the CPUs and the main memory, performance could be reduced even if snapshots are taken on different CPUs.

5.6 Memory Overhead

The size of the snapshot is equal to the heap. When running only one instrument, the memory usage will double. But when several instruments use the same garbage collector, they can share snapshot. The memory usage is

$$atomic\_heap_{size} + heap_{size} * \frac{n+1}{n} \tag{5.3}$$

where n is the number of instruments.

5.7 Sharing Data between Instruments

Making snapshot overhead lighter by dividing data into separate smaller heaps makes it difficult to share data between instruments. Since we don't want to take snapshot of all heaps simultaneously, a garbage collector can not safely trace memory in other heaps.

To avoid freeing objects used by other heaps, a small reference counting collector can be implemented to keep track of references to objects used across instruments. A global register can contain a list of names, addresses, and number of references. In case an instrument is deleted while storing objects referenced by other instruments, deletions of referenced heaps can be delayed. (It may also be possible to create algorithms which only frees parts of memory not used by other instruments.)

For simplicity, we can limit the type of objects which can be accessed across instruments, to globally scoped variables. With this limitation enforced, a further limitation is to only detect at compile time when such objects are referenced from other instruments, so that the amount of code required to updates the register can be minimized.

In case the deletion of heaps sometimes has to be delayed, this scheme causes larger than usual memory usage, but since it's not unpredictable for the user to know the amount of used memory, and the delay doesn't cause higher snapshot time, the scheme does not break realtime performance.

Program 8 Optimized Version of the First Collector

```
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 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
```

6 Basic Implementation

In chapter 5, our garbage collector was described with words and pseudocode. In this chapter we go one step further, and show an almost finished implementation.

The implementation shown here is basic. Later chapters improves the collector by building further on the code in this chapter.

6.1 Structures

```
/*
 * Used when transporting information about allocated memory
 * between collector and mutator.
 */
struct meminfo_minimal:
  start
  size


/*
 * Used by mark-and-sweep.
 * Defines the list element in the list of allocated memory blocks.
 */
struct meminfo:
  next
  start
  end
  marked


/*
 * The main heap structure
 *
 * mheap: Dynamic memory heap
 * audio_function_can_run: A semaphore
 * audio_function: The function processing audio
 * all_meminfos: List of allocated memory blocks
 * ringbuffer: Holds meminfo_minimal objects
 */
struct heap:
  mheap
  audio_function_can_run
  audio_function
  all_meminfos
  ringbuffer
```

6.2 Creating a new Heap

```
1  gc_create_heap(audio_function):
2    heap                        = sys_alloc(sizeof(struct heap))
3    heap.mheap                  = create_mheap(heap_size)
4    heap.audio_function_can_run = SEMAPHORE_INIT(1)
5    heap.audio_function          = audio_function
6    heap.all_meminfos           = {}
7    heap.ringbuffer             = create_ringbuffer(
8                                      MAX_RB_ELEMENTS
9                                      * sizeof(struct meminfo_minimal)
10                                   )
11   return heap
```

The function *create_mheap* on line 3 creates a new heap of dynamic memory. *create_mheap* is described in section 7.1.

6.3 Global Variables

```
heap_size          = 0

roots_start        = NULL
roots_end          = NULL

current_heap       = NULL
snapshot_mem       = NULL
meminfo_pool       = NULL

markandsweep_ready = SEMAPHORE_INIT(0)
snapshot_ready     = SEMAPHORE_INIT(0)

gc_can_run         = false
```

6.4 Initialization

```
gc_init(new_heap_size):
  heap_size    = new_heap_size

  roots_start  = sys_alloc(MAX_ROOTS_SIZE)
  roots_end    = roots_start

  snapshot_mem = sys_alloc(heap_size)
  meminfo_pool = init_pool(sizeof(struct meminfo))

  create_thread(REALTIME_PRIORITY,gc_thread)
  create_thread(REALTIME_PRIORITY,markandsweep_thread)
```

6.5 Allocation

```
1  // Used by mutator
2
3  gc_alloc(heap,size):
4    mem = alloc(heap.mheap, size)
5
6    /* Tell garbage collector about new memory. */
7    ringbuffer_write(
8      heap.ringbuffer,
9      [mem,size]
10   )
11
12   return mem
13
```

The function *alloc* on line 4 allocates memory from the memory heap. *alloc* is described in section 7.1.

Since increased heap size also means increased time taking snapshot, we do not store *size* and a link to the previous allocated memory block before the returned value. Instead, a ringbuffer is used to transport this information to the garbage collector.

6.6 Running the DSP Block Function

(including example of usage)

```
1  // Used by mutator
2
3  run_dsp_block_function(heap):
4
5    //   In case the collector is not finished taking snapshot
6    //   since last time, we have to wait for it.
7    wait(heap.audio_function_can_run)
8
9    //   Run the audio function
10   heap.audio_function(heap)
11
12   //   In case the collector is ready and the collector did not
13   //   run last time, a new collection is started:
14   if is_waiting(snapshot_ready)
15   && is_waiting(markandsweep_ready)
16   && gc_can_run==true:
17      gc_can_run   = false
18      current_heap = heap
19      signal(snapshot_ready)
20   else:
21      gc_can_run = true
22      signal(heap.audio_function_can_run)
23
24 an_audio_function(heap):
25   <create sound>
26
27 main():
28   gc_init(1024*1024) // Heap size
29   heap = gc_create_heap(an_audio_function)
30   start_audio_thread(run_dsp_block_function,heap)
31   sleep(...)
```

gc_can_run is set to *false* after initiating a new collection to prevent two snapshots from being performed in a row.

6.7 The Snapshot Thread

```
// Used by collector

gc_thread():
  while true:
    wait(snapshot_ready)
    heap=current_heap

    reset_roots()
    take_root_snapshot(global_vars_start, global_vars_end)
    take_heap_snapshot(heap)

    num_new_allocations = ringbuffer_read_size(heap.ringbuffer) / sizeof(struct meminfo_minimal)

    signal(heap.audio_function_can_run)

    if num_new_allocations > 0:
       signal(mark_and_sweep_ready)
```

6.8 The Mark-and-Sweep Thread

```
// Used by collector

mark_and_sweep_thread():
  while true:
    wait(mark_and_sweep_ready)
    get_new_meminfos(heap,num_new_allocations)
    run_mark(roots_start,roots_end)
    run_sweep()
```

6.9 Get Memory Info

The function *get_new_meminfos* is called from the mark-and-sweep thread before starting a new collection. *get_new_meminfos* transports newly created memory information from the ringbuffer to the garbage collector's internal list of $meminfos$.

```
// Used by collector

get_new_meminfos(heap,num_new_allocations):
  for i = 0 to num_new_allocations:
      from = ringbuffer_read(
                heap.ringbuffer,
                sizeof(struct meminfo_minimal)
             )

      to        = get_pool(meminfo_pool)

      to.start  = from.start
      to.end    = from.start+from.size
      to.marked = false

      put(to, heap.all_meminfos)
```

6.10 Taking Snapshot

To get a snapshot of the complete environment, both the heap and the roots must be copied.

Heap Snapshot

```
// Used by collector

take_heap_snapshot(heap):
  for i = 0 to heap_size:
      snapshot_mem[i] = heap.mheap[i]
```

Roots Snapshot

```
// Used by collector

reset_roots():
  roots_end = roots_start

take_root_snapshot(start,end):
  size = end - start
  for i = 0 to size:
      roots_end[i] = start[i]
  roots_end += size
```

6.11 Mark

mark traverses all memory recursively and marks all visited blocks by setting the *marked* flag to *true*.

```
1  // Used by collector
2
3  get_snapshot_mempos(heap_mempos):
4    offset = snapshot_mem - current_heap.mheap
5    return heap_mempos + offset
6
7  find_mem(address):
8    foreach mem in current_heap.all_meminfos:
9      if address>=mem.start && address<mem.end:
10        return mem
11   return NULL
12
13 run_mark(start,end):
14   for mempos  = start to end:
15       address = *mempos
16       if address >= current_heap.mheap
17       && address <  current_heap.mheap.freemem:
18          mem = find_mem(address)
19          if mem!=NULL && mem.marked==false:
20             mem.marked = true
21             start      = get_snapshot_mempos(mem.start)
22             end        = get_snapshot_mempos(mem.end)
23             run_mark(start,end)
24
```

The variable *current_heap.mheap.freemem* on line 17 points to the current highest possible address which may contain allocated memory. (The memory heap structure is declared in section 7.1.)

6.12 Sweep

```
1  // Used by collector
2
3  run_sweep():
4    survived_mem={}
5
6    foreach mem in current_heap.all_meminfos:
7      if mem.marked==true:
8         put(mem,survived_mem)
9         mem.marked=false
10     else:
11        size = mem.end-mem.start
12        for i=0 to size:
13            mem.start[i]=0
14        free(current_heap.mheap, mem.start, size)
15        put_pool(meminfo_pool,mem)
16
17   current_heap.all_meminfos=survived_mem
18
```

To avoid later marking memory based on expired pointers, we null out all memory before it can be reused. (Line 12 and 13.) We null out here instead of during allocation to make the mutator run faster.

7 The Dynamic Memory Allocator

In order to allocate memory without risking unpredictable execution time, we need a realtime memory allocator.

The dynamic memory allocator in this chapter uses a minimal algorithm where memory are available from a stack or from a large array of segregated lists.[1] The segregated lists are initially empty, but filled dynamically during execution of the program. When allocating memory, the allocator first tries to pop an element from the start of the most fitting segregated list, and if that list is empty, the stack is used instead.

The allocator uses the simplest type of segregated free lists: One segregated list for every memory size, up to a predefined point. Allocating a larger memory block than the predefined point is impossible. Allocating from a list containing memory blocks of a different size is also impossible.

In 1977, Norman R. Nielsen tested 35 different types of memory allocation algorithms for relatively small simulation programs. A similar looking allocator ("Multiple Free Lists") turned out favorable compared to the other algorithms (such as First fit, Best-fit, and Binary Buddies [WJNB95]), both in terms of effectiveness and memory overhead [Nie77].

For small realtime programs which run in a loop and treat the same events over and over, no fragmentation should occur. In the other end: almost ultimate fragmentation occurs instantly if a program allocates randomly sized blocks.

Both allocating and releasing memory is extremely cheap. Allocation time is even competitive with copying collectors, which only requires a pointer increase [Che70].

In addition, the allocator has a bounded memory usage. The required

[1] List of memory objects of a specific size [WJNB95].

amount of memory is exactly:

$$\sum_{i=0}^{\infty} max_i * i \tag{7.1}$$

where i is the memory block size, and max_i is the highest number of simultaneously allocated i sized blocks.

This makes fragmentation predictable, and for some types of programs, it becomes possible to avoid the fragmentation concerns of non-moving garbage collectors. For instance has [BCR03] recently raised concerns about this issue.

7.1 Basic Implementation

An actual implementation must align the size up to the nearest pointer location, and maybe return NULL or throw an error if there's no memory left. In addition, the number of segregated lists can be reduced by using $\frac{memory\ block\ size}{pointer\ size}$ as array index. But apart from that the algorithm looks like this:

```
struct mheap:
  freemem
  slists

struct mem:
  next

create_mheap(heap_size):
  mheap         = sys_alloc(heap_size)
  mheap.freemem = mheap + sizeof(struct mheap)
  mheap.slists  = sys_alloc(heap_size*sizeof(pointer))

  for i=0 to heap_size:
      mheap.slists[i]=NULL
  for i=0 to heap_size-sizeof(struct mheap):
      mheap.freemem[i]=0

  return mheap

alloc(mheap,size):
  if mheap.slists[size]!=NULL:
     return pop(mheap.slists[size])
  else:
     ret = mheap.freemem
     mheap.freemem+=size
     return ret

free(mheap,start,size):
  put(start,mheap.slists[size])
```

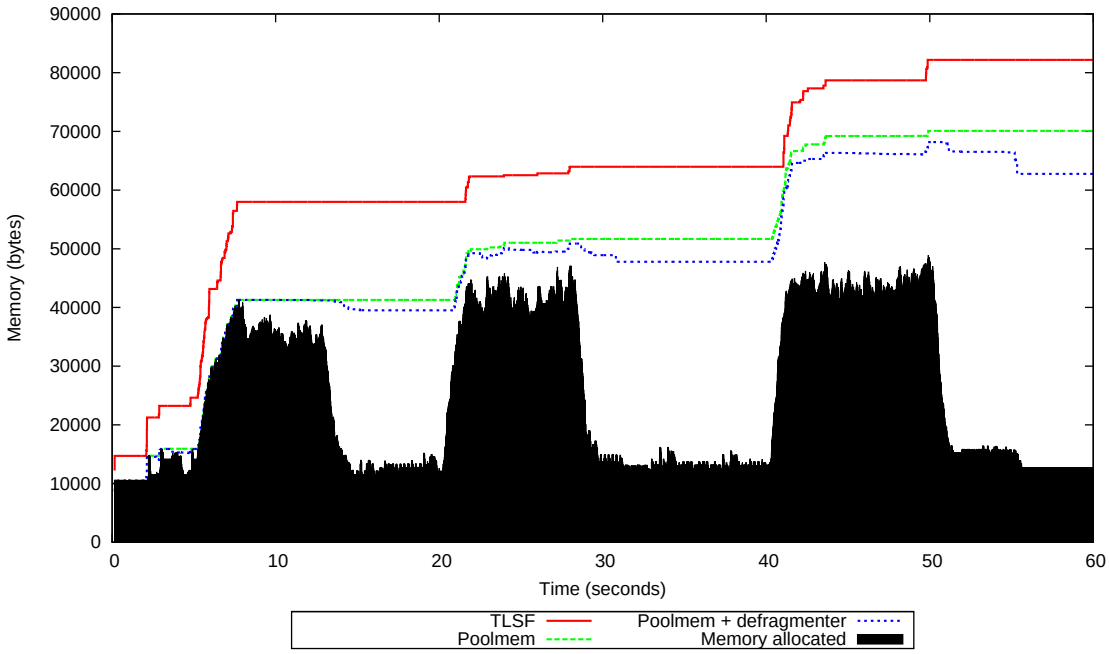

Figure 7.1: Fragmentation of the Pointer Heap

7.2 Fragmentation

The program in appendix A.1 has been used to test fragmentation. The same program is also used to run benchmarks in chapter 11.

The program does not treat the same events over and over, but consists of four parts:

00s-55s: A MIDI synthesizer playing a song. Tones are played using square waves. The synthesizer applies autopanning and ADSR envelope for each tone. Stereo reverb is applied as well.

05s-15s: A triangle wave grain cloud. 500 grains are played.

20s-32s: A synthesized insect swarm based on the "insect" instrument in SND, made by Bill Schottstaedt. 400 insects are played.

40s-50s: A cloud of plucked strings based on the "pluck" instrument in SND, also made by Bill Schottstaedt. The instrument uses the physical model of Jaffe and Smith [JS83]. 2000 strings are plucked.

Playing four different parts, and using several types of synthesize methods, is likely to provoke higher fragmentation.

Figure 7.1 shows the amount of requested memory while running the program, plus the highest allocated memory position in these three memory allocators:

1. *TLSF* is the "Two-Level Segregate Fit" allocator by Miguel Masmano, Ismael Ripoll & Alfons Crespo. TLSF was introduced in section 3.3.

2. *Poolmem* is the allocator described in this chapter

3. *Poolmem + defragmenter* is Poolmem + an attempt to lower fragmentation by running a defragmenter. [2]

The focus is reducing the amount of memory in the memory heap since this reduces time taking snapshots. Therefore, other types of memory used

[2]The defragmenter looks like this:

```
label defragment:
  for each slist in slists:
    for each block in slist:
      if block is at the top of the stack then:
         remove block from slist
         decrease stack
         goto defragment
```

This defragmenter uses a lot of processing power, but it is simple to extend *free* in order to defragment memory equally efficient as running the above code: If the memory block to be freed is on top of the allocator stack, *free* can decrease the stack pointer instead of inserting the block into the correct segregated list. If the memory block placed right before it is also free, and right after the end has a pointer to the next object in the list, the start of the block can be found by following the "prev-pointer" of the next element in the list. (The segrated lists must be doubly linked now). And then the block can be removed from the list, and the stack pointer decreased even further. And so on. (*free* does not require $O(1)$ performance.)

However, the increased use of memory required to implement this (one extra pointer per block) causes more fragmentation than what is gained. At least for the program in appendix A.1. There are also other alternatives, such as using a hash table to store memory headers, but doing so would complicate the code and pollute the CPU cache.

Changing *free* to only decrease the stack pointer for the current object, like this:

```
free(mheap,start,size):
  if start+size == mheap.freemem:
     mheap.freemem = start
  else:
     put(start,mheap.slists[size])
```

...caused no significant improvement in fragmentation. In figure 7.1, the two Poolmem graphs (with and without the extension above) only differed minimally a couple of places, and therefore this graph was not included.

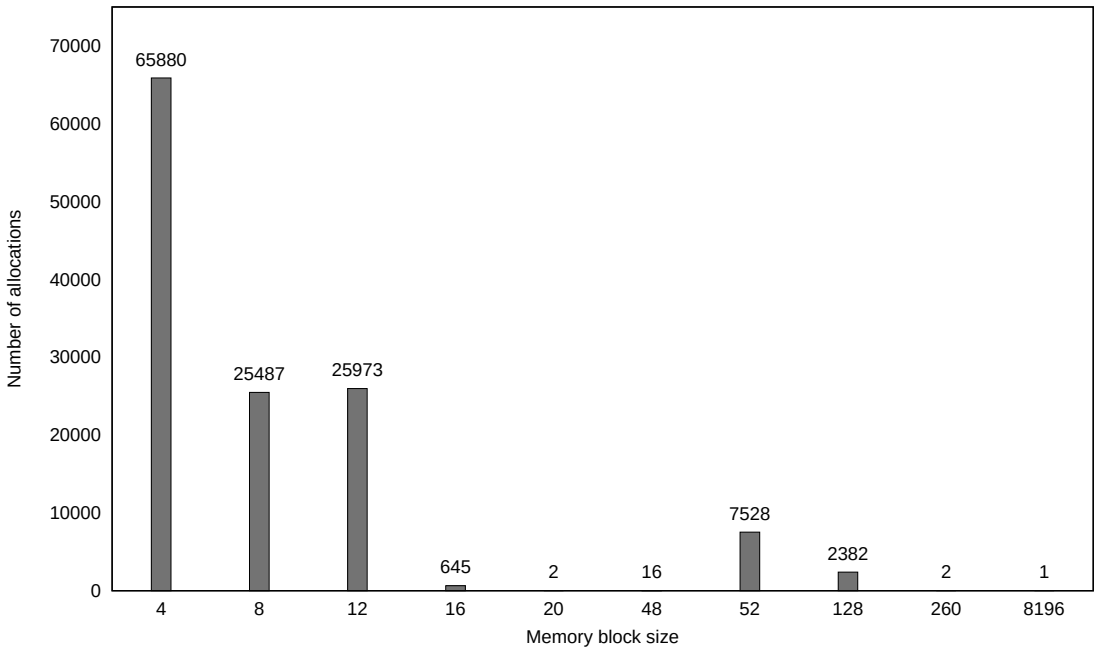

Figure 7.2: Memory block size chart for pointer heap

by the allocators, such as the *mheap.slists* attribute, are not a part of the fragmentation statistics shown in figure 7.1. For the same reason, the main header of the TLSF heap (which is 3180 bytes on a 32 bit computer) is not included in the graphs either, since it's not necessary to take snapshot of it.

7.3 When Poolmem can not be used

Figure 7.3 shows the fragmentation of the heap not containing pointers.[3] We see that the memory is instantly becoming more and more fragmented when the plucked strings start playing. The reason is that the physical model used by the plucked string algorithm allocates a delay line, in which white noise is filtered over and over. The length of the delay line decides the frequency of the string. Since the frequency of each tone is set randomly by the benchmark program, and 2000 tones are played in a few seconds, a large variation in size of allocated memory blocks occurs. A previously unused pool will often be used when a delay line is freed, and this increases fragmentation. We also see that the defragmenter makes no significant difference in this test.

[3]There are two heaps. The heap not containing pointers is not necessary to take snapshot of.

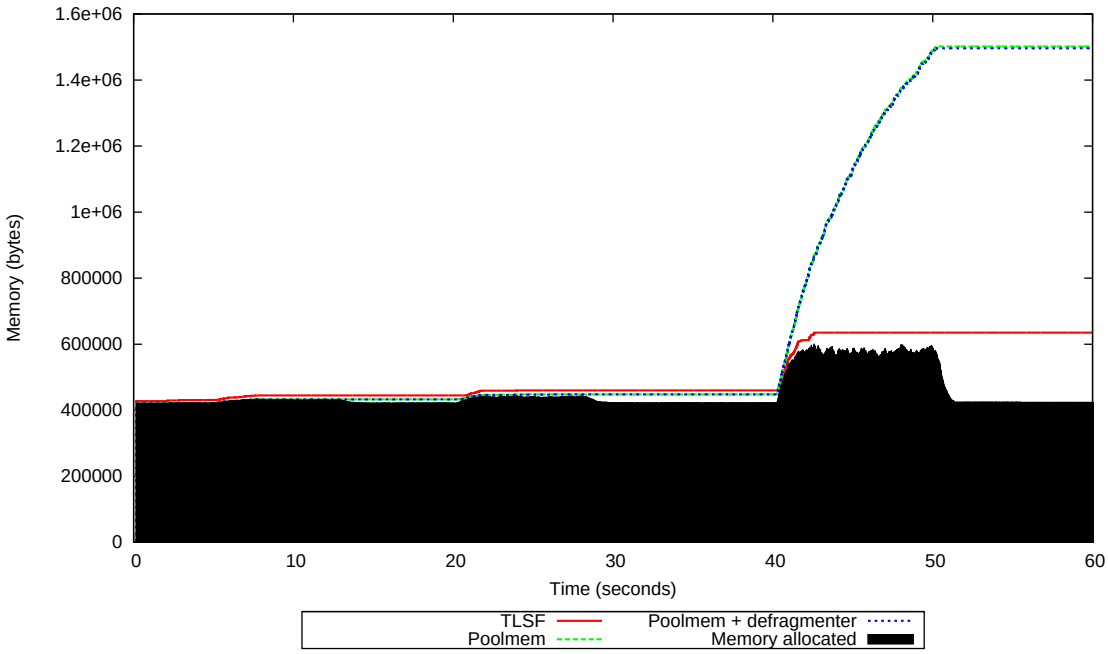

Figure 7.3: Fragmentation of the Atomic Heap

7.4 Thread Safe Implementation

Since *sweep* is running in a parallel thread, *free* and *alloc* can be called simultaneously from two threads. In order to avoid locks, lock-free lists implemented using atomic operations could be used instead of normal single-linked lists [FOL02].

8 Several Heaps and Garbage Collectors

If a program runs several instruments, heap size per instrument can be reduced if each instrument uses its own heap. But the implementation in chapter 6 only handled one heap.

In addition, there were no mechanism to synchronize garbage collectors to avoid more than one snapshot to be taken at the same time. Since the memory bus is the bottleneck when taking snapshots on personal computers, running more than one snapshot simultaneously will increase snapshot duration, even if all snapshots are performed on separate CPUs.

8.1 More than one Heap

The simplest way to support more than one heap is to select beforehand which heap to take garbage collection of. The simplest way to select heap is to use the one with the most allocated memory. The following code implements this functionality,

To avoid running garbage collection again and again on the same heap without any memory being released, a new garbage collection is only run on heaps which have allocated memory since last time a collection was run on it.

```
struct heap:
  mheap
  audio_function_can_run
  audio_function
  all_meminfos
  ringbuffer
  start_new_gc
  next

run_dsp_block_function(heap):
  wait(heap.audio_function_can_run)
  heap.audio_function(heap)
  if heap.start_new_gc==true:
     heap.start_new_gc = false
     current_heap      = heap
     signal(collector_ready)
  else:
     signal(heap.audio_function_can_run)

get_fullest_heap():
  fullest_heap = heaps
  fullest_size = fullest_heap.mheap.freemem - fullest_heap.mheap
  foreach heap in heaps.next:
     used_memory = heap.mheap.freemem - heap.mheap
     if used_memory > fullest_size
     && ringbuffer_read_size(heap.ringbuffer) > 0:
        fullest_heap = heap
        fullest_size = used_memory
  return fullest_heap
```

(code continues on next page)

```
run_dsp_block_functions(heaps):
  if is_waiting(collector_ready)
  && is_waiting(markandsweep_ready)
  && gc_can_run==true:
     gc_heap              = get_fullest_heap()
     gc_heap.start_new_gc = true
     gc_can_run           = false
  else:
     gc_can_run = true
  foreach heap in heaps:
     run_dsp_block_function(heap)

main():
  gc_init(1024*1024) // Heap size
  heap = gc_create_heap(an_audio_function)
  start_audio_thread(run_dsp_block_function,heap)
  start_audio_thread(run_dsp_block_functions,heap)
  sleep(...)
```

8.2 More than one Garbage Collector

Two ways to avoid several snapshots to be taken simultaneously: 1. A server selects which client is allowed to run a garbage collection in the current audio block cycle. 2. A server does the garbage collection, and not the client.

Advantages of running garbage collection on the server:

1. Lower memory overhead: Only one snapshot buffer is needed on the computer.

2. Only one mark-and-sweep process is run simultaneously so that 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 [TS08].

The main disadvantage of running the garbage collection on a server is that it requires large amounts of shared memory. To ensure high performance, shared memory will be required both for the snapshot, roots, and for sending information about allocated memory back and forth. Shared memory is often a limited resource.

Clients Run Garbage Collection on Their Own Heaps

This sections implements the first method: Clients run garbage collection on their own heaps, but connect against a server to synchronize with other clients to avoid several snapshots to be taken simultaneously.

The implementation lets each client run a special listening thread, which, when asked by the server, calculates and sends back the heap usage in permil (‰) of the client's fullest heap. And as usual; only of those heaps which has allocated memory since its last collection.

```
~~gc_can_run = false~~

struct client:
  next
  queue

sync_server_thread():
  clients          = {}
  client_pool      = create_pool(sizeof struct client)
  num_clients      = 0
  gc_can_run       = false
  sq               = create_queue()

  publish_server_queue(sq)

  while true:
    request=receive(sq,sizeof(request))
    switch request:
      case REQUEST_ADD_CLIENT:
           client       = get_pool(client_pool)
           client.queue = receive(sq,sizeof(queue))
           put(client,clients)
           num_clients++
           send(client.queue) // ACK
      case REQUEST_REMOVE_CLIENT:
           queue   = receive(sq,sizeof(queue))
           client  = find_client(clients,queue)
           clients = remove_element(client,clients)
           put_pool(client,client_pool)
           num_clients--
           send(client.queue) // ACK
```

(function continues on next page)

```
case REQUEST_NEW_GC_DECISION:
     client_queue          = receive(sq,sizeof(queue))
     highest_permil        = -1
     fullest_clients_queue = NULL

     if gc_can_run==true:
        foreach client in clients:
          send(client.queue,REQUEST_HEAP_INFO)

        for i = 0 to num_clients:
          permil = receive(sq,sizeof(int))
          if permil > -1:
             queue = receive(sq,sizeof(queue))
             if permil > highest_permil:
                if fullest_clients_queue != NULL:
                   send(fullest_clients_queue,false)
                highest_permil      = permil
                fullest_clients_queue = queue
             else:
                send(queue,false)

        if fullest_clients_queue != NULL:
           send(fullest_clients_queue,true)
           gc_can_run  = false

     else:
        foreach client in clients:
          send_to_client(client.queue,NO_REQUEST)
        gc_can_run  = true

     send(client_queue) // ACK

default:
     error("unknown request")
```

(code continues on next page)

```
server_queue = NULL
client_queue = NULL
gc_decided   = false

heap_decider_thread():
  while true:
    request = receive_from_server(sizeof(request))
    if request==REQUEST_HEAP_INFO:
       if is_waiting(collector_ready):
          gc_heap = find_fullest_heap()
          permil  = 1000 * gc_heap.allocated_mem / heap_size
          send(server_queue,permil,client_queue)
          if receive(client_queue,sizeof(boolean))==true:
             gc_heap.start_new_gc = true
          gc_decided = true
       else:
          gc_decided = true
          send(server_queue,-1)

run_dsp_block_functions(heaps):
  if gc_decided==false:
     send(server_queue,REQUEST_NEW_GC_DECISION,client_queue)
     receive(client_queue)
  foreach heap in heaps:
     run_dsp_block_function(heap)
  gc_decided=false
```

(code continues on next page)

```
sync_init():
  ipc_lock()
    if get_server_queue()==NULL:
       create_thread(REALTIME_PRIORITY,sync_server_thread)
       while get_server_queue()==NULL:
         sleep()
  ipc_unlock()

  server_queue = get_server_queue()
  client_queue = create_queue()

  send(server_queue,REQUEST_ADD_CLIENT,client_queue)

  create_thread(REALTIME_PRIORITY,heap_decider_thread)

gc_init(new_heap_size):
  heap_size    = new_heap_size
  roots_start  = sys_alloc(MAX_ROOTS_SIZE)
  roots_end    = roots_start
  snapshot_mem = sys_alloc(heap_size)
  meminfo_pool = init_pool(sizeof(struct meminfo))
  create_thread(REALTIME_PRIORITY,gc_thread)
  sync_init()
```

In addition, if heap size differs between clients, the server should select the client with the largest heap to run snapshot at least once per second.

The code may also run faster by using interprocess semaphores instead of sending messages. Another optimization is to avoid synchronization if only one client is connected to the server.

Parallel Running Clients

The previous code assumed that all clients connected to the server ran in serial. To handle parallel running clients, interprocess synchronization must be added to the function *run_dsp_block_functions*:

```
run_dsp_block_functions(heaps):
  if gc_decided==false:
    ipc_lock()
       if gc_decided==false:
          send(server_queue,REQUEST_NEW_GC_DECISION,client_queue)
          receive(client_queue)
    ipc_unlock()
  foreach heap in heaps:
     run_dsp_block_function(heap)
  gc_decided=false
```

9 Optimizations and Realtime Adjustments

Realtime audio processing has low upper bound requirements.

9.1 Optimizing Mark

The implementation of *mark* in chapter 6 was inefficient and lacked some precautions to reduce unpredictability.

Avoid Transporting Memory between CPU Caches

Sharing data between the mutator and the collector can cause data to be transported back and forth between two CPU caches while the audio thread is running. Not only takes transportation of data between two CPU caches time by itself, but it's also hard to predict when it happens.

Another related problem is *false sharing*. False sharing means that separate variables placed in memory used by different threads are placed so close to each other that they share CPU cache line [TS08]. When this happens, memory is cached by accident, and we risk two CPUs fighting back and forth over the same cache line data.

Unpredictable cache performance are mostly avoided by restricting frequent access to objects which are used both by the mutator and the collector:

1. The ringbuffer is replaced by a "transport stack" (see Appendix A.2). The transport stack contains two stacks. The mutator and the collector are not allowed to access the same stack simultaneously.

2. Access to global variables and the heap object are restricted in the collector by allocating a new memory block (called *mark_variables*) to

store heap variables and global variables. This memory block is then used exclusively by the garbage collector thread.

```
struct heap:
  mheap
  audio_function_can_run
  audio_function
  all_meminfos
  tr_stack
  start_new_gc
  next
  writer

meminfo_pool        = NULL

struct mark_variables:
  all_meminfos
  num_new_allocations
  heap_start
  heap_end
  snapshot_mem_offset
  meminfo_pool
  reader
```

(code continues on next page)

```
gc_alloc(heap,size):
  minimal = stack_write(heap.writer,sizeof(struct meminfo_minimal))
  minimal.start = alloc(heap.mheap,size)
  minimal.size  = size
  return minimal.start

get_new_meminfos( mv ,num_new_allocations):
  for i = 0 to num_new_allocations
      from = stack_read (
                 mv.reader ,
                 sizeof(struct meminfo_minimal)
             )
      to   = get_pool( mv. meminfo_pool)

      to.start  = from.start
      to.end    = from.start+from.size
      to.marked = false

      put(to, mv. all_meminfos)

prepare_new_gc(mv, heap):
  mv.all_meminfos         = heap.all_meminfos
  mv.heap_start           = heap.mheap
  mv.heap_end             = heap.mheap.freemem
  mv.snapshot_mem_offset  = snapshot_mem - heap.mheap

  tr_stack_switch(heap.tr_stack)
  heap.writer             = heap.tr_stack.writer
  mv.reader               = heap.tr_stack.reader
```

(code continues on next page)

```
mark_and_sweep_thread():
  while true:
    mv = wait(mark_and_sweep_ready)
    get_new_meminfos(heap, mv.num_new_allocations)
    run_mark(mv, roots_start, roots_end)
    run_sweep()

gc_thread():
  mv              = sys_alloc(sizeof(struct mark_variables))
  mv.meminfo_pool = init_pool(sizeof(struct meminfo))

  while true:
     wait(collector_ready)
       heap = current_heap
       reset_roots()
       take_root_snapshot(global_vars_start, global_vars_end)
       take_heap_snapshot(heap)
       prepare_new_gc(mv, heap)
       mv.num_new_allocations = stack_space(mv.reader) / sizeof(struct meminfo_minimal)
     signal(heap.audio_function_can_run)

     if mv.num_new_allocations > 0:
        signal(mark_and_sweep_ready , mv)

get_snapshot_mempos(mv,heap_mempos):
  return heap_mempos + mv.snapshot_mem_offset

find_mem( mv,address):
  foreach mem in mv.all_meminfos:
    if address>=mem.start && address<mem.end:
       return mem
  return NULL
```

(code continues on next page)

```
run_mark(mv,start,end):
  for mempos  = start to end:
      address = *mempos
      if address >= mv.heap_start
      && address <  mv.heap_end:
         mem = find_mem(mv,address)
         if mem!=NULL && mem.marked==false:
            mem.marked = true
            start      = get_snapshot_mempos(mv,mem.start)
            end        = get_snapshot_mempos(mv,mem.end)
            run_mark(mv,start,end)

gc_create_heap(audio_function):
  heap                         = sys_alloc(sizeof(struct heap))
  heap.mheap                   = create_mheap(heap_size)
  heap.audio_function_can_run = SEMAPHORE_INIT(1)
  heap.audio_function          = audio_function
  heap.all_meminfos            = {}

  heap.tr_stack                = tr_stack_create(
                                   MAX_RB_ELEMENTS
                                   * sizeof(struct meminfo_minimal)
                                 )
  heap.writer                  = heap.tr_stack.writer
  return heap


gc_init(new_heap_size):
  heap_size    = new_heap_size
  roots_start  = sys_alloc(MAX_ROOTS_SIZE)
  roots_end    = roots_start
  snapshot_mem = sys_alloc(heap_size)
  meminfo_pool = init_pool(sizeof(struct meminfo))
  create_thread(REALTIME_PRIORITY,gc_thread)
  create_thread(REALTIME_PRIORITY,markandsweep_thread)
```

Avoid Running Out of Transport Stack

In order to keep code clean, there was an important precaution which was not implemented in the previous code.

The duration between each time a new garbage collection is started will vary, and if garbage collection takes too long time while *gc_alloc* is called too many times, the transport stack would be full. *gc_alloc* would then fail since the mutator are not allowed to allocate system memory. This problem is solved by making sure the transport stack is switched and emptied after processing every block. Then we can guarantee an upper bound on the number of allocations allowed to make between each block.

Optimizing find_mem

The efficiency of *mark* was $O(m * n)$, where m is the total amount of allocated memory, and n is the number of allocations. A significant amount of time is likely to be spent in the $find\_mem$ function, since it only has an efficiency of $O(n)$ and is called every time a potential pointer to the heap is found. Three quick ways to optimize $find\_mem$ are:

1. Pointers are likely to point directly to the start of allocated memory blocks (i.e. not pointing inside objects (*interior pointers*)). *find_mem* will run faster if it can look up a hash table instead of iterating through the list of $meminfos$.

2. Similarly, to avoid iterating the list of *meminfo*s every time a potential interior pointer is found, interior pointers are also inserted into the hash table (when they are found). In order to remove interior pointers from the hash table when the corresponding *meminfo* object is freed, the interior pointer must also be stored somewhere in the *meminfo* object. A list of interior pointers is added to each *meminfo* object.

3. When a *false address* is found (a value found during trace that points somewhere inside the heap, but does not point to, or inside, any live memory object), this false address is also inserted into the hash table. Additionally, since these false addresses must be removed from the hash table after each collection (an action which is not necessary for interior pointers), they are inserted into a special linked list as well so that they can easily be found and deleted when *mark-and-sweep* is finished.

   Although false pointers should be rare, the overhead of doing this is always low, while the consequence of not doing it could be high.

If interior pointers are not required to be recognized, both point 2 and 3 are unnecessary, while for frequent use of new interior pointers, a binary search tree (such as a splay tree) could perform better than a hash table.

```
struct mark_variables:
  all_meminfos
  num_new_allocations
  heap_start
  heap_end
  snapshot_mem_offset
  meminfo_pool
  reader
  hash_table
  false_addresses
  address_pool


struct meminfo:
  next
  start
  end
  marked
  interiors


struct address:
  next
  address
```

(code continues on next page)

```
// find_mem is now mostly rewritten:
find_mem(mv,address):
  mem=get_hash(mv.hash_table,address)
  if mem!=-1: // get_hash returns -1 when the address is not found
     return mem

  // Needs some memory for storing the address in the hash table.
  add=get_pool(mv.address_pool)
  add.address=address

  // Maybe it's an interior pointer...
  foreach mem in mv.all_meminfos:
     if address>=mem.start && address<mem.end:
        // Yes, it's an interior pointer
        put_hash(mv.hash_table, address, mem)
        put(add,mem.interiors)
        return mem

  // No, it was a false address! [1]
  put_hash(mv.hash_table, address, NULL)
  put(add, mv.false_addresses)
  return NULL

// [1] This is probably very unlikely to happen.
```

(code continues on next page)

```
get_new_meminfos(mv,num_new_allocations):
  for i = 0 to num_new_allocations
      from = stack_read(
                 mv.reader,
                 sizeof(struct meminfo_minimal)
             )
      to   = get_pool(mv.meminfo_pool)

      to.start     = from.start
      to.end       = from.start+from.size
      to.marked    = false
      to.interiors = {}

      put(to, mv.all_meminfos)
      put_hash(mv.hash_table, to.start, to)

run_sweep( mv ):
  survived_mem={}
  foreach mem in mv .all_meminfos:
    if mem.marked==true:
       put(mem,survived_mem)
       mem.marked=false
    else:
       size = mem.end-mem.start
       for i=0 to size:
           mem.start[i]=0
       free(current_heap.mheap, mem.start, size)
       remove_hash(mv.hash_table,mem.start)
       foreach interior in mem.interiors:
          remove_hash(mv.hash_table,interior.address)
          put_pool(mv.address_pool,interior)
       put_pool( mv .meminfo_pool,mem)
  current_heap.all_meminfos=survived_mem

  // The false addresses must be cleared between each
  // collection.
  foreach add in mv.false_addresses:
    remove_hash(mv.hash_table,address)
    put_pool(mv.address_pool,add)
  mv.false_addresses={}
```

9.2 Optimizing Sweep

Similar to *mark*, both real and false CPU cache sharing can cause degraded and unpredictable performance in *sweep*.

To avoid this, *sweep* should avoid calling $free()$, which both reads and writes to the heap. Instead, we transport memory information from the collector to the mutator on an array of $minimal\_meminfo$ objects. (The variable *sweep_mem* holds this array.)

One place the mutator could free memory, is before calling *alloc*. One thought could be that if the mutator tries to free more memory than it is asked to allocate, we should not run out of memory.[1] However, since freeing memory is less important than processing audio, it would be preferable to use a lower priority thread for freeing memory to avoid stealing time from code having to reach a deadline. It must also be specified that this thread must run on the same CPU as the audio thread to avoid CPU cache misses and false sharing.

```
struct heap:
  mheap
  audio_function_can_run
  audio_function
  all_meminfos
  tr_stack
  start_new_gc
  next
  writer
  ilock
```

```
run_free              = SEMAPHORE_INIT(0)
free_thread_is_ready  = SEMAPHORE_INIT(0)
sweep_mem             = NULL
sweep_mem_size        = 0
```

(code continues on next page)

[1] For the minimal memory allocator in section 7, which uses segregated lists extensively, this is not always true.

```
free_thread():
  while true:

    signal(free_thread_is_ready)
    heap = wait(run_free)

    ilock1_lock(heap.ilock)
      for i=0 to sweep_mem_size:
          minimal = sweep_mem[i]
          mem     = minimal.mem
          size    = minimal.size
          for i1=0 to size step 64:
              ilock1_pause(heap.ilock)
              for i2=i1 to min(size,i1+64):
                  mem[i2]=0
          free(heap.mheap,mem,size)
    ilock1_unlock(heap.ilock)


transport_meminfos(mv, free_mem):
  sm     = sweep_mem
  sm_pos = 0

  wait(free_thread_is_ready)

  foreach mem in free_mem:
     if sm_pos == MAX_FREE_ELEMENTS:
        sweep_mem_size = sm_pos
        signal(run_free,mv.heap)
        wait(free_thread_is_ready)
        sm_pos = 0
     minimal=sm[sm_pos++]
     minimal.mem=mem.start
     minimal.size=mem.end-mem.start

  sweep_mem_size = sm_pos
  signal(run_free,mv.heap)
```

(code continues on next page)

```
// Mostly rewritten
run_sweep(mv):
  free_mem     = {}
  survived_mem = {}

  foreach mem in mv.all_meminfos:
      if mem.marked==true:
         put(mem,survived_mem)
         mem.marked=false
      else:
         put(mem,free_mem)
  current_heap.all_meminfos=survived_mem

  transport_meminfos(mv,free_mem)

  foreach mem in free_mem:
     remove_hash(mv.hash_table,mem.start)
     foreach interior in mem.interiors:
       remove_hash(mv.hash_table,interior.address)
       put_pool(mv.address_pool,interior)
     put_pool(mv.meminfo_pool,mem)

  foreach address in mv.false_addresses:
    remove_hash(mv.hash_table,address)
  mv.false_addresses={}


run_dsp_block_function(heap):
  wait(heap.audio_function_can_run)

  set_priority(free_thread,audio_thread_priority) // To avoid priority inversion!
  ilock2_lock(heap.ilock)
    heap.audio_function(heap)
  ilock2_unlock(heap.ilock)
  set_priority(free_thread,free_thread_priority)

  if heap.start_new_gc==true:
     heap.start_new_gc = false
     current_heap      = heap
     signal(collector_ready)
  else:
     signal(heap.audio_function_can_run)
```

(code continues on next page)

```
gc_create_heap(audio_function):
  heap                         = sys_alloc(sizeof(struct heap))
  heap.mheap                   = create_mheap(heap_size)
  heap.audio_function_can_run = SEMAPHORE_INIT(1)
  heap.audio_function          = audio_function
  heap.all_meminfos            = {}
  heap.tr_stack                = tr_stack_create(
                                   MAX_RB_ELEMENTS
                                   * sizeof(struct meminfo_minimal)
                                 )
  heap.writer                  = heap.tr_stack.writer
  heap.ilock                   = create_ilock()
  return heap

gc_init(new_heap_size):
  heap_size    = new_heap_size
  roots_start  = sys_alloc(MAX_ROOTS_SIZE)
  roots_end    = roots_start
  snapshot_mem = sys_alloc(heap_size)
  meminfo_pool = init_pool(sizeof(struct meminfo))

  sweep_mem = sys_alloc(sizeof(struct meminfo_minimal)*MAX_FREE)

  gc_thread   = create_thread(REALTIME_PRIORITY,gc_thread)
  free_thread = create_thread(REALTIME_PRIORITY,free_thread)

  cpunum = get_cpu_num(audio_thread)
  run_on_cpu(!cpunum, gc_thread)
  run_on_cpu(cpunum,  sweep_free_thread)
```

The code above uses a lock to prevent the audio thread and the free thread from running simultaneously. Since the free thread frequently calls a pause operation (*ilock1_pause*) in order to avoid blocking the audio thread, an extended type of lock is used. This lock is implemented in appendix A.3.

Furthermore, since $free$ and *alloc* are now protected by locks from running simultaneously, it is not necessary to run a thread safe version of the dynamic memory allocator.

9.3 Priorities and Scheduling Policies

All threads used by the garbage collector needs to run with a realtime scheduling policy to avoid running out of memory.

A GUI process could for instance delay garbage from being reclaimed if the collector runs with incorrect priority and scheduling policy.

1. To prevent mark-and-sweep from blocking the audio thread, mark-and-sweep must run with lower priority than the audio thread. 2. Since the audio thread waits for snapshot to finish, the snapshot thread must have at least the same priority as the audio thread.

In a POSIX environment [POS90], SCHED_FIFO and SCHED_RR are the two realtime scheduling policies. The difference between SCHED_FIFO and SCHED_RR is that SCHED_FIFO never gives up execution unless another realtime process with higher priority is ready. SCHED_RR is quite similar, but will give up execution after a short while in case another realtime process with the same priority is ready [Har93].

Below is a list of priorities and scheduling policies for our collector when running in a POSIX environment.

```
x                              = audio thread priority
snapshot thread priority       = SCHED_FIFO / x
sync server thread priority    = SCHED_FIFO / x
heap decider thread priority   = SCHED_FIFO / x
mark+sweep thread priority     = SCHED_RR   / 0
free thread priority           = SCHED_RR   / 0
```

In Linux distributions, the JACK [LFO05] audio thread priority is normally set somewhere between 10% and 60% of maximum SCHED_FIFO priority. Audio thread priority in other systems are likely to be in the same range.

9.4 Atomic Memory

Atomic memory is memory not containing pointers to the heap. The most common type of atomic memory in audio programs are sample buffers.

It is not necessary to scan atomic memory for pointers in the *mark* phase, and it is not necessary to take snapshot of atomic memory.

To optimize the collector for atomic memory, we use two heaps, and let the

user of the collector specify when memory is atomic. BDW-GC distinguishes atomic and non-atomic memory by providing two functions for allocating memory: *GC_malloc_atomic*() and *GC_alloc*(). Rollendurchmesserzeitsammler provides similar functions called *tar_alloc_atomic* and *tar_alloc*. (The implementation has been left out.)

9.5 Optimizing Snapshot

So far, all memory in the heap has been copied when taking snapshot. (*full snapshot*.) To lower CPU usage, only the used parts of the heap can be copied instead (*partial snapshot*). But a full snapshot must still be taken once per second. If not, there wouldn't be a lower bound, only an upper bound.

The effect in a realtime system is to trade CPU time, previously spent copying unused memory, for time which can now be spent doing non-realtime operations such as updating graphics or handling user events:

```
take_heap_snapshot(heap):
  if at_least_one_second_since_last_full_copy(heap):
      size = heap_size
  else:
      size = heap.mheap.freemem - heap.mheap

  for i = 0 to size:
      snapshot_mem[i] = heap.mheap[i]
```

Another optimization is performed by checking if memory has been allocated in the *run_dsp_block_function* function:

```
run_dsp_block_function(heap):
  wait(heap.audio_function_can_run)

  set_priority(free_thread,audio_thread_priority)
  ilock2_lock(heap.ilock)
    heap.audio_function(heap)
  ilock2_unlock(heap.ilock)
  set_priority(free_thread,free_thread_priority)

  if heap.start_new_gc==true
  && (stack_space(heap.writer) > 0
       || at_least_one_second_since_last_full_copy(heap)):
     heap.start_new_gc = false
     current_heap      = heap
     signal(collector_ready)
  else:
     signal(heap.audio_function_can_run)
```

Sleeping instead of Taking Full Snapshot

An alternative is to measure the time it takes to take a full snapshot, and then sleep for the remaining time after taking a partial snapshot. This has at least two advantages:

1. The OS can let other programs run while the snapshot thread is sleeping.

2. If using too much CPU, the glitch in sound is heard immediately, not after about a second.

However, sleeping instead of taking full snapshot requires that the operating system can sleep accurately for a fraction of the time it takes to play a block of audio. In other words, sleeping needs to work for durations of at least a magnitude lower than 1.33 milliseconds in order to properly support all soundcard configurations. Such fine grained sleeping are not commonly supported by normal operating systems, and on top of that comes latency to wake up a processes, which is limited by hardware.[2]

[2]Scheduling latency has been reported to be 0.1 milliseconds at least on one laptop running a Linux operating system tweaked for realtime performance [Ble10].

9.6 Safety Buffer

As discussed in section 1.5, a conservative garbage collector is not guaranteed to always use the same amount of memory every time a program is run.

To compensate for this variation, we need a safety buffer. If the program needs to use the safety buffer, we give the user a warning:

```
struct heap:
  mheap
  audio_function_can_run
  audio_function
  all_meminfos
  tr_stack
  start_new_gc
  next
  writer
  ilock
  too_much_memory_warning


run_dsp_block_function(heap):
  wait(heap.audio_function_can_run)

  set_priority(free_thread,audio_thread_priority)
  ilock2_lock(heap.ilock)
    heap.audio_function(heap)
  ilock2_unlock(heap.ilock)
  set_priority(free_thread,free_thread_priority)

  if heap.too_much_memory_warning == false
  && (heap.mheap.freemem - heap.mheap) > minimal_full_snapshot_size():
      heap.too_much_memory_warning = true
      warning("Using safety buffer. Program may run out of memory.")

  if heap.start_new_gc==true:
  && (stack_space(heap.writer) > 0
       || at_least_one_second_since_last_full_copy(heap)):
     heap.start_new_gc = false
     current_heap      = heap
     signal(collector_ready)
  else:
     signal(heap.audio_function_can_run)
```

The above code calls the function *minimal_full_snapshot_size()* which returns the size of the heap subtracted by the size of the safety buffer.

The size of the safety buffer should be so large that if there is no warning during rehearsal, the program should not run out of memory during a concert.

Fragmentation, false pointers and memory not reclaimed fast enough can make the available amount of memory unpredictable.

In chapter 7, we analyzed fragmentation both for the custom memory allocator (Poolmem), and TLSF. As long as we don't allocate randomly sized blocks, Poolmem seems to perform excellently, providing virtually no fragmentation. Almost the same can be said about TLSF, but TLSF does not provide predictable fragmentation.

A safety buffer does not have to compensate for fragmentation per se, but rather for *unexpected* fragmentation. If the fragmentation is similar every time a program is run, it behaves predictably, and no safety buffer is needed. (Although a warning should probably still be given if there's little memory left.)

In [MRR+08], an extensive test of fragmentation for TLSF is performed, and the author claims worst-case fragmentation to be lower than 30%. Since expected fragmentation is likely to be much higher (perhaps even magnitudes higher) than unexpected fragmentation, we assume that we are never going to see more than a 30% worst-case unexpected fragmentation. Considering that humans also should get a warning in case there is little memory left (even if memory usage is predictable), using 30% of the heap for safety sounds like a proper value:

```
minimal_full_snapshot_size():
  return (heap_size * 7) / 10
```

10 Finding Snapshot Duration

We want to achieve consistent snapshot performance. We also want to take snapshots often enough to avoid running out of memory.

Defining the problem

There are two problems that can cause inconsistent snapshot performance:

1. Cache misses, wake up latency for threads, having to wait for other threads to yield execution, higher-priority threads taking over execution in the middle of a snapshot, hardware interrupts, and so forth, can occasionally cause snapshot to take longer time.

2. The memory bus is shared by all CPUs. Non-realtime programs have equal access to the memory bus if they run on a different CPU.

The first problem is a list of factors that only happens occasionally (at least for realtime programs), while the second problem can happen if other simultaneously running programs are accessing memory. We try to solve the problems like this:

1. To account for the first problem, we set up a window in which a full snapshot must be taken. If we are inside the window, and there was not enough time to take a full snapshot this block, we try again at the next block. In case we don't manage to take a full snapshot within the window, a warning is given. We set the window size to 1 second.[1]

[1] As usual, 1 second is used since it seems like a fairly appropriate value. In an implementation, however, it would be natural for the user to be able to adjust it.

2. To account for the second problem, we try to find a minimum snapshot performance (e.g. in megabytes per milliseconds). We find minimum snapshot performance by dividing best case performance by the number of CPUs.

Setting up a Hypothesis

We define a hypothesis H which says that for every heap size, each heap having a sequence of measurements within a one second window, at least one snapshot will perform better than $\frac{1}{c}$ of best-case performance:

$$H : \forall m : \exists\, t \in B : \quad p(m, t, c-1) \geq \frac{p(M)}{c}, \quad t \leq 1 \tag{10.1}$$

where c is the number of CPUs, $B = \{b,\ 3b,\ 5b,\ 7b,\ \ldots\}$, b is the audio block duration (in seconds), $p(M)$ is best-case performance, M is a number much larger than the size of the CPU cache, and $p(m, t, c-1)$ is the snapshot performance when copying an m-sized heap at time t while $c-1$ other CPUs access the memory bus simultaneously.

Testing the Hypothesis

We try to verify H by running several tests:

- $p(m, t, 0)$ is found by running the program in appendix A.4, using test_num==0. This tests snapshot performance while no other threads access the memory bus simultaneously. This test shows typical snapshot performance when no other application is running at the same time. $p(m, t, 0)$ is not used to verify H, but it is included to better verify that the test has been properly set up.

- $p(m, t, c-1)$ is found by running the program in appendix A.4, using test_num==1. This tests snapshot performance while 40 other non-realtime threads access the memory bus simultaneously. (The benchmark thread always runs with realtime priority)

- $p(m, t, c-1)_{RT}$ is found by running the program in appendix A.4, using test_num==2. This tests snapshot performance while $c-1$ other realtime threads access the memory bus simultaneously. The benchmark thread runs on CPU number 0, the first realtime thread runs on CPU number 1, and so on.

- $p(M)$ is found by running the program in appendix A.5. The program in appendix A.5 runs several tests under various conditions, and returns the fastest result of all tests.

Test setup

- CPU: Intel Core i7 920 @ 2.67GHz. CPU has 4 cores. Each core has a 32kB L1 data cache and a 256kB L2 cache. An 8 MB L3 cache is shared by all cores. Hyper threading, CPU scaling, and Turbo Mode is disabled.
- Memory: DDR3 1066MHz.
- Operating Systems:
  - 32 bit Linux Fedora Core 11, running the 2.6.29.6-1.rt23.4.fc11.ccrma.i586.rt PREEMPT_RT realtime kernel [Kac10].
  - 64 bit Linux Fedora Core 13, running the 2.6.33 kernel.
- All tests measures the time it takes to take snapshot of 38 different heap sizes: 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5, 10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0, 50.0, 55.0, 60.0, 65.0, 70.0, 75.0, 80.0, 85.0, 90.0, 95.0, and 100.0 MB.
- Each heap is tested repeatedly for 1 second.
- Raw benchmark data is found in appendix B.1.

Data analysis

Best-case results are shown in figure 10.1 and figure 10.2. We see from the figures that neither $p(m, t, c-1)_{RT}$ nor $p(m, t, c-1)$ are below $\frac{p(M)}{c}$ for any heap size. H seems to be correct.

10.1 Implementation

Following the hypothesis H, we need a one second window in which a full snapshot must be taken. The required snapshot performance within that window must be better than $\frac{p(M)}{c}$ at least once. As soon as the required snapshot performance has been met, we close the window.

If we spend more than a second trying to take full snapshot, we give up, make sure that at least a partial snapshot is taken (so that a new collection can start), and give the user a warning.

This scheme should work since it seems impossible for the system to be almost continuously tortured for more than a second without the user

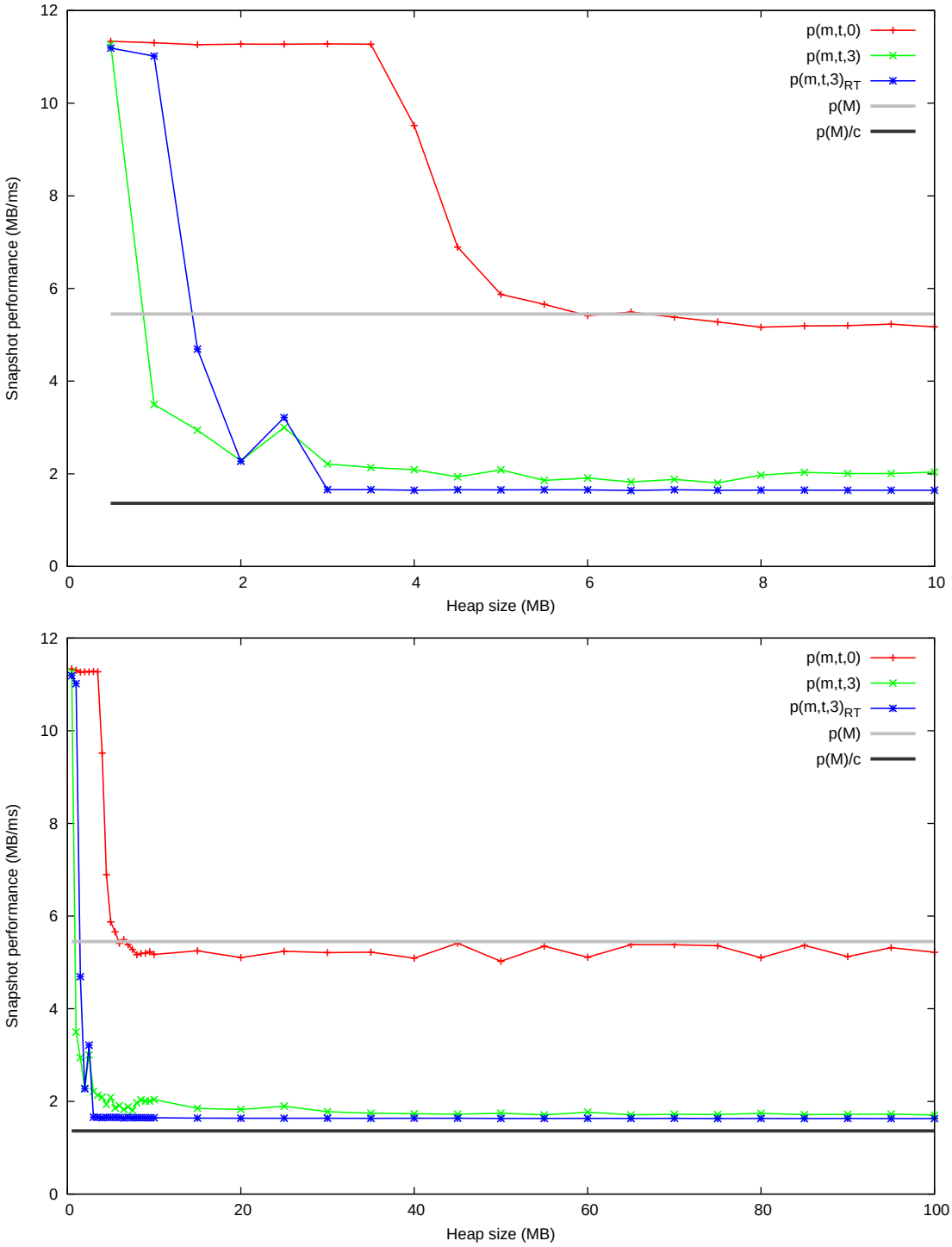

Figure 10.1: Snapshot Performance on a 32 Bit Linux Realtime Kernel

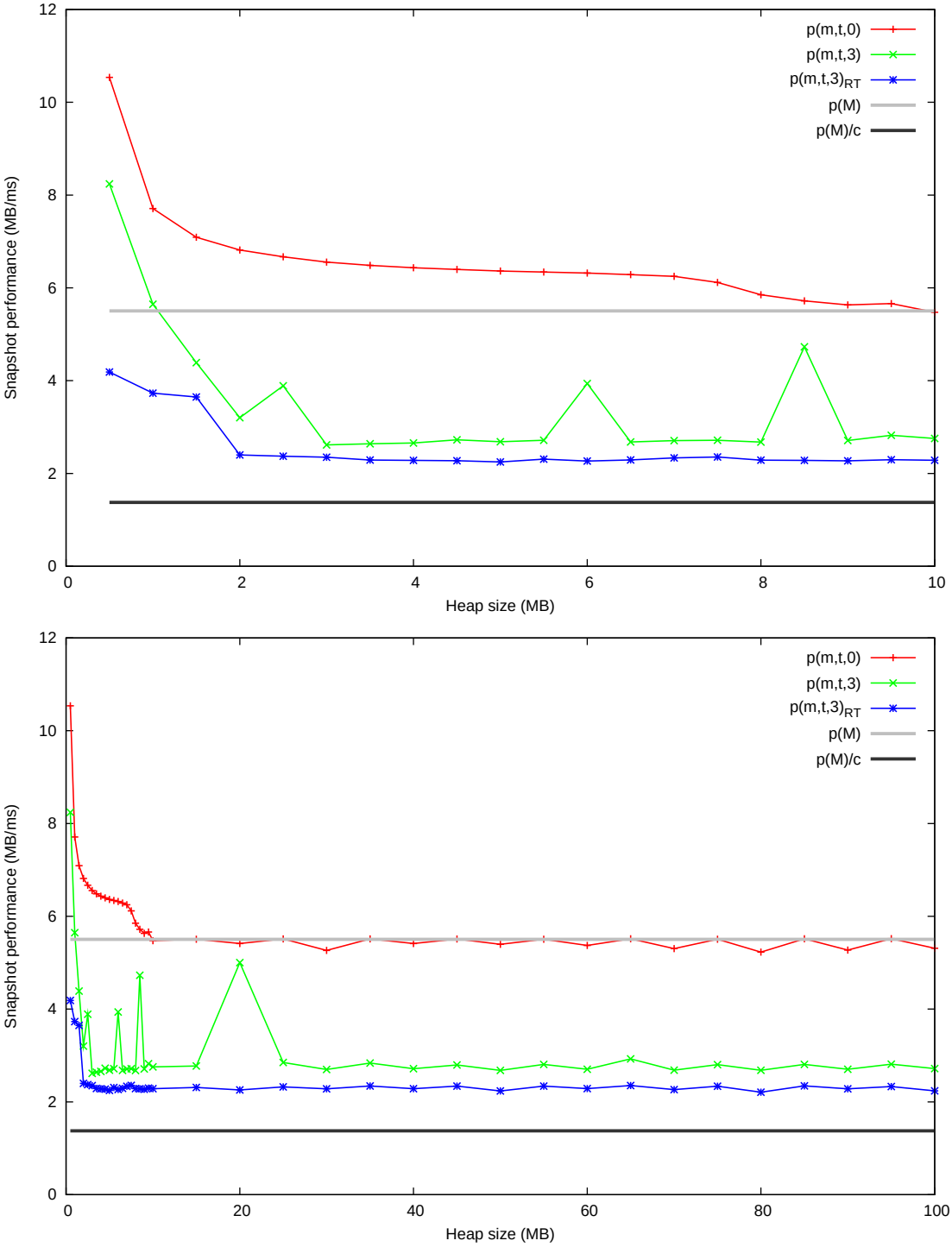

Figure 10.2: Snapshot Performance on a 64 Bit Linux Kernel

running a program which either has a bug, or where this behavior could not be predicted or at least discovered. If a snapshot takes longer time than usual, the user should get a warning long before the program starts using so much memory that a partial snapshot can not be taken within the deadline.

Instead of dividing ideal snapshot duration by c to create an upper bound on snapshot duration, we increase the ratio of the heap reserved as safety buffer. Although more memory is required this way, the extra memory is not wasted since it will be available for safety, plus that we reduce the risk of having to busy loop (see code):

```
ideal_snapshot_performance = read_from_file()  /* p(M), found at compile time. */
full_snapshot_duration     = heap_size/ideal_snapshot_performance
last_full_snapshot         = 0
window_length              = 1 second

minimal_full_snapshot_size():
  if num_cpu==1:
     return (heap_size*7)/10
  else:
     return heap_size/num_cpu

take_heap_snapshot(heap): /* Rewritten: */
  time = get_time()

  if time < last_full_snapshot + 1 second:
     return take_heap_snapshot_before_window(heap)
  if time < last_full_snapshot + 1 second + window_length:
     return take_heap_snapshot_inside_window(heap)
  else
     return take_heap_snapshot_after_window(heap)
```

(code continues on next page)

```
take_heap_snapshot_before_window(heap):
  pos          = 0
  partial_size = heap.mheap.freemem - heap.mheap
  start_time   = get_time()

  while true:
    next_pos = pos + 16 * 1024
    if next_pos > partial_size:
       next_pos = partial_size
    while pos < next_pos:
       snapshot_mem[pos] = heap.mheap[pos]
       pos++

    duration          = start_time - get_time()

    if pos==partial_size:
       return true
    elseif duration >= full_snapshot_duration:
       return false


take_heap_snapshot_inside_window(heap):
  pos          = 0
  size         = heap_size
  partial_size = heap.mheap.freemem - heap.mheap
  start_time   = get_time()

  while true:
    next_pos = pos + 16 * 1024
    if next_pos > size:
       next_pos = size
    while pos < next_pos:
       snapshot_mem[pos] = heap.mheap[pos]
       pos++

    time              = get_time()
    duration          = start_time - time

    if pos==size:
       /* force worst-case by busy-looping */
       while (start_time - get_time()) < full_snapshot_duration:
         do nothing
       last_full_snapshot = time
       return true
    elseif duration > full_snapshot_duration:
       return pos > partial_size
```

(code continues on next page)

```
take_heap_snapshot_after_window(heap): /* This should not happen. */
  pos          = 0
  size         = heap_size
  partial_size = heap.mheap.freemem - heap.mheap
  start_time   = get_time()

  while true:
    next_pos = pos + 16 * 1024
    if next_pos > size:
       next_pos = size
    while pos < next_pos:
       snapshot_mem[pos] = heap.mheap[pos]
       pos++

    time              = get_time()
    duration          = start_time - time

    if pos==size:
       last_full_snapshot = time
       return true
    elseif duration > full_snapshot_duration
    && pos >=  partial_size:
       warning("Unable to take full snapshot.")
       last_full_snapshot = time /* Not true, but a warning has been given. */
       return true


gc_thread():
  mv              = sys_alloc(sizeof(struct mark_variables))
  mv.meminfo_pool = init_pool(sizeof(struct meminfo))

  while true:
     wait(collector_ready)
       heap = current_heap
       reset_roots()
       take_root_snapshot(global_vars_start, global_vars_end)
       new_collection_can_run = take_heap_snapshot(heap)
       prepare_new_gc(mv, heap)
       mv.num_new_allocations = stack_space(mv.reader) / sizeof(struct meminfo_minimal)
     signal(heap.audio_function_can_run)

     if mv.num_new_allocations > 0:
        if new_collection_can_run:
          signal(mark_and_sweep_ready, mv)
        else:
          get_new_meminfos(heap,mv.num_new_allocations)
```

11 Benchmarks

In these benchmarks, the results of Rollendurchmesserzeitsammler (Rollendurch) is compared with the results of a traditional mark-and-sweep collector. The mark-and-sweep collector was made by modifying Rollendurch to work directly on the heap and not on a snapshot.[1]

The most important comparison is whether taking snapshots is more efficient and more predictable than running *mark* in a mark-and-sweep collector. The reason for comparing snapshot with *mark*, is that these are the two operations in their respective collectors that can not run simultaneously with the mutator.

For C programs which uses normal pointers and are not providing read or write barriers, no other type of garbage collector can work. The only exception is the snapshot-at-the-beginning collector described in section 1.6, which uses copy-on-write to create the snapshot. But a snapshot-at-the-beginning collector using copy-on-write can not provide better worst-case performance than Rollendurch.

11.1 Setup

- The program in appendix A.1 was used to run the test. The same program was also used to run the fragmentation test in chapter 7, and a detailed description of the program is found in section 7.2.

  Both Rollendurch and the mark-and-sweep collector used the memory allocator described in chapter 7.

- The same computer which was used in chapter 10 is also used here, a

[1] Compile time option: NO_SNAPSHOT.

2.67GHz Intel Core i7 920 processor with 8 MB shared L3 cache. Hyper threading, CPU scaling and Turbo mode was disabled.

- Time values were gathered using the CPU time stamp counter (TSC), and all threads were running with a realtime scheduling policy, either SCHED_FIFO or SCHED_RR.

- All threads were locked to run on only one CPU to prevent timings from fluctuating. (Rollendurch bounds all threads to one CPU by default.) The threads were configured like this:

Mark-and-sweep		Rollendurch	
Thread	CPU #	Thread	CPU #
Audio Thread	0	Audio Thread	0
Free Thread	0	Free Thread	0
Mark+Sweep Thread	0	Mark+Sweep Thread	1
		Snapshot Thread	1

Table 11.1: CPU Assignments

- The size of the pointer-containing heap was 4MB. 70724 bytes were used at most by the program. (i.e. the highest allocated memory position), and the size of the roots were 18584 bytes.

- The size of the non-pointer-containing heap was set to 128MB, but the size of the non-pointer-containing heap should not have any affect on the performance.

- All tests were ran 5 times for both types of garbage collector. The worst results of the 5 tests were used in this chapter. The difference between the 5 tests were minimal for both collectors, except for worst-case blocking time for the mark-and-sweep collector, which differed by a factor of 1.78,

- A new garbage collection was started at each second block. But only if new memory had been allocated since last collection. Then the collection would be delayed at least one block. Block duration during this test was 5.8ms, which is a normal setting.

- All benchmark data are listed in appendix B.2.

11.2 Results

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	234752	1024	3912	4096	60
audio (ms)	10264.081	0.387	0.989	3.154	10375
free (ms)	20.637	0.000	0.000	0.004	243225
sweep (ms)	30.093	0.000	0.003	0.023	10374
mark (ms)	358.098	0.021	0.069	0.219	5187
roots snapshot (ms)	9.308	0.001	0.002	0.028	5187
partial snapshot (ms)	23.157	0.001	0.005	0.012	5128
full snapshot (ms)	43.182	0.712	0.732	0.741	59
total (ms)	10748.557				

Table 11.2: Rollendurch. Benchmarks

	Total	Minimum	Average	Maximum	#
audio (ms)	10270.766	0.386	0.990	3.168	10374
free (ms)	20.500	0.000	0.000	0.004	243228
sweep (ms)	28.433	0.000	0.003	0.017	10376
mark (ms)	376.456	0.026	0.073	0.317	5188
total (ms)	10696.154				

Table 11.3: Mark-and-Sweep Benchmarks

Table 11.2 and table 11.3 shows the worst benchmarks among the results of the two respective collectors. We are going to use these data. (The same data are also found in figure B.6 and figure B.9 in appendix B.2.)

Blocking Time

	Mark-and-sweep			Rollendurch		
	Min.	Avg.	Max.	Min.	Avg.	Max.
mark	0.026ms	0.073ms	0.317ms			
roots snapshot				0.001ms	0.002ms	0.028ms
partial snapshot				0.001ms	0.005ms	0.012ms
total	0.026ms	0.073ms	0.317ms	0.002ms	0.006ms	0.040ms

Table 11.4: Blocking Time

Table 11.4 shows the amount of time the garbage collector blocks audio processing during one audio block. Full snapshot is not included in this table.

Since the roots were only 18584 bytes long, the maximum value 0.28ms could be caused by cache misses.

Relative Time

Mark-and-sweep		Rollendurch	
Blocking	Non-blocking	Blocking	Non-blocking
3.52%	0.46%	0.70%	3.80%

Table 11.5: Time Spent by the Garbage Collector. ($\frac{time}{time_{audio+gc}}$)

Although Rollendurch uses a little bit more CPU than the mark-and-sweep collector, most of it is non-blocking and spent running in parallel on a different CPU than audio processing.

Worst Case

Mark-and-sweep			Rollendurch		
Min	Avg.	Max	Min	Avg.	Max
n/a	n/a	n/a	0.712ms	0.732ms	0.741ms

Table 11.6: Time Simulating Worst Case

Table 11.6 shows the time simulating worst case. For Rollendurch, this is the time taking full snapshot. The numbers approximately show the time it would take to take partial snapshot if 3912kB of memory was used by the program.

11.3 Data analysis

Simulating worst-case for mark-and-sweep has not been implemented since we don't known how. (It's most likely impossible to do reliably.) But if we compare total time running *mark* in mark-and-sweep with the the total time taking partial snapshot in Rollendurch ($\frac{376.455780ms}{23.157463ms}$), worst-case for mark-and-sweep is about 16 times higher than Rollendurch.

Comparing worst-case occurrences (instead of combined time) indicates that worst-case for mark-and-sweep is about 26 times higher than Rollendurch ($\frac{0.317276ms}{0.012048ms}$).

These two attempts to find worst-case were speculative. But we do know from the data that the highest blocking time for a normal mark-and-sweep collector was quite large (0.3ms), even though only 70724 bytes were used. Taking partial snapshot never took more than 0.01ms in any of the tests. Higher number of cache misses in *mark* can explain some of the difference.

12 The Second Garbage Collector

This chapter describes a novel concurrent non-moving realtime garbage collector. The collector uses a minimal write barrier to support very large heaps and also to avoid the problems described in section 5.3.

In this collector, only the roots are copied between blocks. Copying the roots should normally be a light operation and the size of the roots should not vary during execution.

Furthermore, as we saw in section 1.2, pointers are never written inside DSP loops, and therefore the additional number of instructions caused by using a write barrier should be low,

12.1 Description of the Algorithm

Similar to the first garbage collector, this collector also runs a mark-and-sweep on a snapshot. The snapshot and the heap are properly separated. The snapshot is only accessed by the collector, and the heap is only accessed by the mutator.

When the collector works on a snapshot instead of the heap, we avoid synchronizations between the collector and the mutator, and the CPU cache used by the mutator is not cooled down, which could contribute to unpredictable performance.

The write barrier writes to a special block of memory we call a *complementary block*. The complementary block has the same size as the heap and the snapshot, and contains data which are different between the heap and the snapshot.

The write barrier is simple. It only stores the *R-value* into the same position in the complementary block, as in the heap. This operation should

Program 9 C Macro for Writing a Pointer to the Heap

```
#define WRITE_HEAP_POINTER(L_VALUE,R_VALUE)do{       \
  void* _gc_pointer = (void*)R_VALUE;                \
  char* _gc_mempos  = (char*)L_VALUE;                \
  *((void**)(_gc_mempos+snap_offset)) = _gc_pointer; \
  *((void**)_gc_mempos)               = _gc_pointer; \
}while(0)
```

normally only require one or two instructions.[1] (But a few extra instructions are needed if it's necessary to check whether the *L-Value* is in the heap.)

Two complementary blocks are used to store newly written heap-pointers. None of them are accessed simultaneously by the collector and the mutator, hence cache conflicts are avoided. When we start a new garbage collection, the two complementary blocks swap position so that *comp_block_gc* becomes *comp_block_wb* and vice versa. The collector then starts merging new data into the snapshot by scanning the *comp_block_gc* complementary block. (This happens in the function *unload_and_reset_comp_block* in program 10). Scanning the complementary block happens in parallel, so it should not disrupt audio. The scanning should be cheap compared to mark-and-sweep, and especially compared to audio processing.

Similar to the first garbage collector, this collector can also be optimized by using two heaps. One heap for pointer data, and another heap for atomic data. Write barriers and complementary blocks are not needed for atomic data.

12.2 Complementary Stacks

The complementary blocks are scanned for pointers in the function *unload_and_reset_comp_block* in program 10. This could put a strain on the memory bus. (Although far less than the strain caused by taking full snapshots in the first collector).

To avoid this, we can extend the write barrier to store complementary block addresses in a stack. This way, the collector will know the exact positions which has been modified since last collection. If the stack becomes full, the complementary block has to be scanned manually, as before. Although the stack should not be too large, a warning could be given in case the stack becomes full, so that the default size can be increased. In the meantime, a larger stack can be allocated automatically.

[1] Two x86 instructions were generated by the Gnu C compiler for the write barrier.

Program 10 Basic Version of the Second Collector

```
ps                    = sizeof(void*)

heap_pointers         = calloc(ps,FULL_HEAP_SIZE)

snapshot              = calloc(ps,FULL_HEAP_SIZE)
comp_block_wb         = calloc(ps,FULL_HEAP_SIZE)
comp_block_gc         = calloc(ps,FULL_HEAP_SIZE)

snap_offset           = comp_block_gc - heap_pointers

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
      comp_block_gc[i] = UNUSED
      comp_block_wb[i] = UNUSED

unload_and_reset_comp_block()
  for i in all live memory do
      if comp_block_gc[i] != UNUSED then
         snapshot[i]      = comp_block_gc[i]
         comp_block_gc[i] = UNUSED
      endif

gc_thread()
  loop forever
    wait for collector semaphore
    unload_and_reset_comp_block()
    run mark and sweep on snapshot

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

This is likely to work well since a new collection is initiated as often as possible, and therefore the write barrier is likely to be used only a few times between each collection. This solution should both reduce access to the memory bus and lower CPU usage by the collector. Thus, it could be worth the cost of having a slightly less cheap write barrier, as shown in program 11.

Contrary to the first write barrier, this write barrier does not work for multithreaded programs. But by using thread-local complimentary stacks, we avoid that problem.

Program 12

Program 12 extends the collector to use complementary stacks. Program 12 also adds another thread so that roots can be copied concurrently with programs waiting for the *audio function* to finish. Reallocating stacks happens in the *mark-and-sweep thread*, which does not block audio processing.

Additional buffer

A required optimization when adding complementary stacks, is to add an additional buffer between the complementary stacks and the complementary blocks. This buffer lowers the chance for the collector having to call *unload_and_reset_comp_block*, but more importantly is that it limits the required size of the stack. A small stack causes less memory to be accessed from the mutator (which could cause cache misses), and it also decreases the chance of cooling the CPU cache. (It's better to cool the CPU cache of the collector)

The buffer should be large enough for the complementary stacks to be emptied between each audio block, at least as long as nothing extreme happens.

The additional buffer is not implemented in program 12.

12.3 Memory Overhead

The snapshot can not be shared between several instruments since the content of the snapshot in the previous collection is also used in the current. One of the complementary blocks can however be shared since it's only used while initiating a new garbage collection. Therefore, the memory usage is

$$atomic\_heap_{size} + heap_{size} * \frac{n * 3 + 1}{n} \tag{12.1}$$

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

Program 11 Extended C Macro for Writing a Pointer to the Heap

```
#define WRITE_HEAP_POINTER(L_VALUE,R_VALUE)do{          \
  void* _gc_pointer       = (void*)R_VALUE;             \
  char* _gc_mempos        = (char*)L_VALUE;             \
  char* _gc_comp_mempos     = _gc_mempos+snap_offset;    \
  if(comp_stack_wb < comp_stack_wb_top)                    \
    *(comp_stack_wb++) = _gc_comp_mempos;                  \
  *((void**) _gc_comp_mempos )          = _gc_pointer;    \
  *((void**) _gc_mempos)              = _gc_pointer;    \
}while(0)
```

12.4 Write Barrier for the Roots

By using a write barrier for the roots, we avoid copying the roots before starting a new collection. This lowers pause times further, but increases write barrier impact and puts more pressure on complementary stacks.

However, since the roots are copied concurrently between audio blocks, a write barrier for the roots would probably never make any difference for audio processing.

Program 12 Extended Version of the Second Collector

```
comp_stack_wb_bot = calloc(ps,DEFAULT_STACK_SIZE)
comp_stack_gc_bot = calloc(ps,DEFAULT_STACK_SIZE)
comp_stack_wb     = comp_stack_wb
comp_stack_gc     = comp_stack_gc
comp_stack_wb_top = comp_stack_wb + DEFAULT_STACK_SIZE
comp_stack_gc_top = comp_stack_gc + DEFAULT_STACK_SIZE

stack_size      = DEFAULT_STACK_SIZE
stack_size_gc   = DEFAULT_STACK_SIZE
stack_size_wb   = DEFAULT_STACK_SIZE

unload_from_stack()
  for i = 0 to comp_stack_gc-comp_stack_gc_bot do
      pos                = comp_stack_gc_bot[i]
      snapshot[pos]      = comp_block_gc[pos]
      comp_block_gc[pos] = UNUSED
  comp_stack_gc = comp_stack_gc_bot
```

(code continues on next page)

```
update_snapshot()
  if comp_stack_gc < comp_stack_gc_top then
     unload_from_stack()
  else
     if stack_size == stack_size_gc && stack_size<MAX_STACK_SIZE then
        warning("stack full")
        stack_size = stack_size * 2
     endif
     unload_and_reset_comp_block() // back-up solution
  endif

mark_and_sweep_thread()
  loop forever
    wait for mark_and_sweep semaphore
    run mark and sweep on snapshot
    if stack_size_gc < stack_size then
       comp_stack_gc_bot = realloc(comp_stack_gc_bot, stack_size)
       comp_stack_gc     = comp_stack_gc_bot
       comp_stack_gc_top = comp_stack_bot + stack_size

swap()
  swap(&comp_block_gc,&comp_block_wb)
  snap_offset = comp_block_wb - heap_pointers
  swap(&comp_stack_gc,&comp_stack_wb)
  swap(&comp_stack_gc_top,&comp_stack_wb_top)
  swap(&comp_stack_gc_bot,&comp_stack_wb_bot)
  swap(&stack_size_gc,&stack_size_wb)

gc_thread()
  loop forever
    wait for collector semaphore
    if mark_and_sweep is waiting then
       swap()
       if there might be garbage then
          copy roots to roots_snapshot
          signal audio semaphore
          update_snapshot()
          signal mark_and_sweep
       else
          signal audio semaphore
       endif
    else
       signal audio semaphore
    endif

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

13 Conclusion

Besides describing two garbage collectors, this thesis has also discussed typical features of audio code, memory allocation and fragmentation, plus issues with CPU cache and the memory bus. Its focus has been realtime audio processing on multiprocessor personal computers.

13.1 Achievements

1. Described and implemented a conservative garbage collector for realtime audio processing which can replace Hans Boehm's conservative collector for C and C++ in existing programming languages. Benchmarks shows that the collector is almost as efficient as a mark-and-sweep collector, but with significantly lower pause times. The collector simulates worst-case once per second to provide predictable performance.

2. Described a new type of garbage collector for audio processing which can provide minimal pause times.

13.2 Future Work

1. Investigate further how to make snapshot performance faster by preventing non-realtime threads from using the memory bus while taking snapshot. This is a problem which goes further than copying memory. Any realtime process accessing memory outside the CPU cache can experience unpredictable performance since non-realtime processes have equal access to the memory bus. One way to achieve this would

be to modify the Linux kernel scheduler to suspend all non-realtime processes when a realtime process is running.

2. Implement and test the second garbage collector. Quantitative evaluations are needed to verify the design choices in Chapter 12. In addition, the collector should be tested against other concurrent realtime collectors, such as the Metronome collector or the STOPLESS collector.

3. Find how to decide whether audio code spent more time than usual during the last block to avoid taking snapshot at that block. In addition, investigate advantages and disadvantages of letting the mutator stop the snapshot process instead of waiting for the previous snapshot to finish.

4. Implement a server for bounding realtime audio processes to CPUs and properly schedule optional realtime operations such as taking snapshots. This functionality can be implemented by extending the JACK audio system.

13.3 Beyond Audio Processing

The two garbage collectors are not limited to audio processing.

The First Garbage Collector

If future personal computers continue to have more and faster memory, the first garbage collector becomes more relevant also for other types of usages. The benchmarks in chapter 11 showed that Rollendurchmesserzeitsammler only used a little bit more CPU than the mark-and-sweep collector. Since most of the time was spent running in parallel with the mutator, both pause times and execution time were lowered. This indicates that programs using normal mark-and-sweep collectors can run faster, and significantly lower pause times, by taking snapshot of the heap. The price is a higher memory overhead.

The Second Garbage Collector

Although the second collector has a very large memory overhead, it should work well also for other types of usages than audio processing. By using a write barrier for the roots, it should be possible to achieve pause times in the range of nanoseconds. If the current trend of personal computers continues,

where the increase in memory is larger than the increase in CPU speed, the second garbage collector will become more relevant.

A Source Codes

A.1 Benchmark Program

```
(define-stalin (Reverb delay-length)
  (Seq (all-pass :feedback -0.7 :feedforward 0.7 :size 1051)
       (all-pass :feedback -0.7 :feedforward 0.7 :size 337)
       (all-pass :feedback -0.7 :feedforward 0.7 :size 113)
       (Sum (comb :scaler 0.742 :size 9601)
            (comb :scaler 0.733 :size 10007)
            (comb :scaler 0.715 :size 10799)
            (comb :scaler 0.697 :size 11597))
       (delay :size delay-length:-s)))

(define-stalin (Stereo-pan c)
  (define gakk 9)
  (Split Identity
         (* left)
         (* right)
    :where left  (* sqrt2/2 (+ (cos angle) (sin angle)))
    :where right (* sqrt2/2 (- (cos angle) (sin angle)))
    :where angle (- pi/4 (* c pi/2))
    :where sqrt2/2 (/ (sqrt 2) 2)))

(define-stalin (Fx-ctrl clean wet Fx)
  (Sum (* clean)
       (Seq Fx
            (* wet))))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (gen-sound :while (-> adsr is-running)
        (Seq (Prod (square-wave :frequency (midi->hz (midi-note)))
                   (midi-vol)
                   (-> adsr next))
             (Stereo-pan (/ (midi-note) 127))))
```

```
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (-> adsr stop)))
      :where adsr (make-adsr :a 20:-ms
                             :d 40:-ms
                             :s 0.2
                             :r 10:-ms))))

(<rt-stalin> :runtime-checks #f
 (seed 500)

 ;; triangle-wave grain cloud
 (spawn
   (wait 5:-s)
   (range i 1 500
     (wait (random 30):-ms)
     (define osc (make-triangle-wave :frequency (between 50 2000)))
     (define dur (between 400 2000):-ms)
     (define e (make-env '((0 0)(.5 .05)(1 0)) :dur dur))
     (sound :dur dur
       (out (* (env e) (triangle-wave osc))))))

 ;; Insect swarm, algorithm implemented by Bill Schottstaedt.
 (spawn
   (wait 20:-s)
   (range i 1 400
     (wait (between 0.01:-s 0.03:-s))
     (spawn
       (define dur (between 0.4 1.6))
       (define frequency (between 600 8000))
       (define amplitude (between 0.03 0.2))
       (define amp-env '(0 0 25 1 75 .7 100 0))
       (define mod-freq (between 30 80))
       (define mod-skew (between -30.0 -2.0))
       (define mod-freq-env '(0 0 40 1 95 1 100 .5))
       (define mod-index (between 250 700.0))
       (define mod-index-env '(0 1 25 .7 75 .78 100 1))
       (define fm-index (between 0.003 0.700))
       (define fm-ratio .500)

       (define degree 45.0)
       (define distance 1.0)
       (define reverb-amount 0.005)
       (let* ((carrier (make-oscil :frequency frequency))
              (fm1-osc (make-oscil :frequency mod-freq))
              (fm2-osc (make-oscil :frequency (* fm-ratio frequency)))
              (ampf (make-env amp-env :scaler amplitude
                                      :duration dur))
              (indf (make-env mod-index-env :scaler (hz->radians mod-index)
                                            :duration dur))
              (modfrqf (make-env mod-freq-env :scaler (hz->radians mod-skew)
                                              :duration dur))
              (fm2-amp (hz->radians (* fm-index fm-ratio frequency))))
```

```
        (sound :dur dur:-s
          (let* ((garble-in (* (env indf)
                               (oscil fm1-osc (env modfrqf))))
                 (garble-out (* fm2-amp (oscil fm2-osc garble-in))))
            (out (* (env ampf)
                    (oscil carrier (+ garble-out garble-in))))))))))

;; pluck strings (jaffe-smith algorithm, implemented by Bill Schottstaedt)
(spawn
  (wait 40:-s)
  (range i 1 2000
    (wait (random 0.01):-s)
    (define dur (between 0.1 1.2))
    (define freq (between 50 400))
    (define amp (between 0.3 0.8))
    (define weighting (between 0.2 0.5))
    (define lossfact (between 0.9 0.99))

    (define (getOptimumC S o p)
      (let* ((pa (* (/ 1.0 o) (atan (* S (sin o)) (+ (- 1.0 S) (* S (cos o))))))
             (tmpInt (inexact->exact (floor (- p pa))))
             (pc (- p pa tmpInt)))
        (if (< pc .1)
            (do ()
                ((>= pc .1))
              (set! tmpInt (- tmpInt 1))
              (set! pc (+ pc 1.0))))
        (list tmpInt (/ (- (sin o) (sin (* o pc))) (sin (+ o (* o pc)))))))

    (define (tuneIt f s1)
      (let* ((p (/ (mus-srate) f)) ;period as float
             (s (if (= s1 0.0) 0.5 s1))
             (o (hz->radians f))
             (vals (getOptimumC s o p))
             (T1 (car vals))
             (C1 (cadr vals))
             (vals1 (getOptimumC (- 1.0 s) o p))
             (T2 (car vals1))
             (C2 (cadr vals1)))
        (if (and (not (= s .5))
                 (< (abs C1) (abs C2)))
            (list (- 1.0 s) C1 T1)
            (list s C2 T2))))

    (let* ((vals (tuneIt freq weighting))
           (wt0 (car vals))
           (c (cadr vals))
           (dlen (caddr vals))
           (lf (if (= lossfact 0.0) 1.0 (min 1.0 lossfact)))
           (wt (if (= wt0 0.0) 0.5 (min 1.0 wt0)))
           (tab (make-vct dlen))
           ;; get initial waveform in "tab" -- here we can introduce 0's to simulate
           ;; different pick positions, and so on -- see the CMJ article for
```

```
            ;; numerous extensions.  The normal case is to load it with white
            ;; noise (between -1 and 1).
            (allp (make-one-zero (* lf (- 1.0 wt)) (* lf wt)))
            (feedb (make-one-zero c 1.0)) ;or (feedb (make-one-zero 1.0 c))
            (ctr 0))

       (range i 0 dlen
         (vct-set! tab i (- 1.0 (random 2.0))))

       (sound :dur dur:-s
         (let ((val (vct-ref tab ctr))) ;current output value
           (vct-set! tab ctr (* (- 1.0 c)
                                (one-zero feedb
                                          (one-zero allp val))))
           (set! ctr (+ ctr 1))
           (if (>= ctr dlen) (set! ctr 0))
           (out (* amp val)))))))


 ;; Midi synthesizer (plays in the background all the time)
 (gen-sound
   (Seq (In (softsynth))
        (Par (Fx-ctrl 0.3 0.04 (Reverb 0.03))
             (Fx-ctrl 0.3 0.04 (Reverb 0.31))))))


(define midi-filename "/gammelhd/gammelhd/home/kjetil/mus220c/sinclair.MID")

(begin
  (system (<-> "aplaymidi --port=129 " midi-filename ""))
  (sleep 3) ;; wait for reverb and decay
  (rte-silence!)
  (tar_bench_print (get_last_rt_heap))
  (tar_end_fragmentation_analysis (get_last_rt_heap)))
```

A.2 Transport Stack

```
struct stack:
  start
  curr
  end

struct tr_stack:
  reader
  writer
  size

tr_stack_create(size):
  tr_stack              = sys_alloc(sizeof(struct tr_stack_owner))
  tr_stack.size         = size

  tr_stack.writer       = sys_alloc(sizeof(struct stack))
  tr_stack.reader       = sys_alloc(sizeof(struct stack))

  tr_stack.writer.start = sys_alloc(size)
  tr_stack.writer.curr  = tr_stack.writer.start
  tr_stack.writer.end   = tr_stack.writer.curr + size

  tr_stack.reader.start = sys_alloc(size)
  tr_stack.reader.curr  = tr_stack.reader.start
  tr_stack.reader.end   = tr_stack.reader.start // (nothing to read yet)

  return tr_stack

// Must not be called unless stack_space(tr_stack.reader)==0
tr_stack_switch_buffer(tr_stack):
  temp                  = tr_stack.writer
  tr_stack.writer       = tr_stack.reader
  tr_stack.reader       = temp

  tr_stack.reader.end   = tr_stack.reader.curr
  tr_stack.reader.curr  = tr_stack.reader.start

  tr_stack.writer.end   = tr_stack.writer.start + tr_stack.size
  tr_stack.writer.curr  = tr_stack.writer.start
```

(code continues on next page)

```
stack_write(stack,size):
  ret        = stack.pos
  stack.pos += size
  return ret

// (Exactly similar to stack_write)
stack_read(stack,size):
  ret        = stack.pos
  stack.pos += size
  return ret

stack_space(stack):
  return stack.end-arb.stack
```

A.3 Interruptible Locks

```
struct ilock:
  lock
  please_pause
  is_pausing
  go

create_ilock():
  ilock              = sys_alloc(sizeof(struct ilock))
  ilock.lock         = LOCK_INIT
  ilock.is_pausing   = false
  ilock.please_pause = false
  ilock.go           = SEMAPHORE_INIT(0)

ilock1_lock(ilock):
  lock(ilock.lock)

ilock1_unlock(ilock):
  ilock.is_pausing=false
  unlock(ilock.lock)

ilock1_pause(ilock):
  if ilock.please_pause==true:
     ilock.is_pausing=true
     unlock(ilock.lock)
     wait(ilock.go)
     lock(ilock.lock)

ilock2_lock(ilock):
  ilock.please_pause=true
  lock(ilock.lock)

ilock2_unlock(ilock):
  ilock.please_pause=false
  unlock(ilock.lock)
  if ilock.is_pausing==true:
     signal(ilock.go)
```

A.4 A program for finding $p(m, t, c)$

```
#define CPU_MHZ 2667


#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <pthread.h>
#include <sched.h>



  /********************/
 /* Global variables */
/********************/
static double program_start;
static int heap_size;
static int test_num;
static int num_disturbance_threads;
static double busy_lengths[NUM_TESTS] = {
  0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5,
  5.0, 5.5, 6.0, 6.5, 7.0, 7.5, 8.0, 8.5, 9.0, 9.5,
  10.0, 15.0, 20.0, 25.0, 30.0, 35.0, 40.0, 45.0,
  50.0, 55.0, 60.0, 65.0, 70.0, 75.0, 80.0, 85.0, 90.0, 95.0, 100.0
};



  /********************/
 /*    Statistics    */
/********************/
#define MAX_NUM_SAMPLES (1024*1024*10)
#define NUM_TESTS 38

static double stats[MAX_NUM_SAMPLES];
static int stat_num=0;

static int double_compare(const void *a, const void *b){
  const double *da = a;
  const double *db = b;
  return *da  > *db;
}

static void get_statistics(double *min,double *q1,double *median,double *q3,double *max){
  int n=stat_num;
  qsort(stats,n,sizeof(double),double_compare);
  *min    = stats[0];
  *q1     = stats[(int)((n+1)/4)]; //approx enough
  *median = stats[(int)((n+1)/2)]; //approx enough
  *q3     = stats[(int)(3*(n+1)/4)]; //approx enough
  *max    = stats[n-1];
}
```

```
static void add_sample(double sample){
  stats[stat_num++]=sample;
}

static void reset_statistics(void){
  memset(stats,0,MAX_NUM_SAMPLES*sizeof(double));
  stat_num=0;
}



  /*********************/
 /* Threads and timing */
/*********************/
static void bound_thread_to_cpu(int cpu){
  cpu_set_t set;
  CPU_ZERO(&set);
  CPU_SET(cpu,&set);
  pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &set);
}

static int set_pthread_priority(pthread_t pthread,int policy,int priority){
  struct sched_param par={0};
  par.sched_priority=priority;

  if ((pthread_setschedparam(pthread, policy, &par)) != 0) {
    abort();
    return 0;
  }
  return 1;
}

static void set_realtime(int type, int priority){
  set_pthread_priority(pthread_self(),type,priority);
}

// Copied from the JACK source. (http://www.jackaudio.org)
typedef int _Atomic_word;
static inline _Atomic_word __attribute__ ((__unused__))
__exchange_and_add(volatile _Atomic_word* __mem, int __val)
{
  register _Atomic_word __result;
  __asm__ __volatile__ ("lock; xaddl %0,%1"
                        : "=r" (__result), "=m" (*__mem)
                        : "0" (__val), "m" (*__mem));
  return __result;
}

static inline int atomic_add_get_oldval(int *mem,int inc){
  return __exchange_and_add(mem,inc); //
}


#if defined(__i386__)
// Also copied from the JACK source
typedef unsigned long long cycles_t;
static inline cycles_t get_cycles (void){
  unsigned long long ret;
  __asm__ __volatile__("rdtsc" : "=A" (ret));
  return ret;
}
#endif
```

```
static double get_ms(void){
#if defined(__x86_64)
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  return ts.tv_sec * 1000.0 + ((double)ts.tv_nsec) / 1000000.0;
#else
  return ((double)get_cycles()) / (1000.0*(double)CPU_MHZ);
#endif
}


// clear memory manually to make sure the memory is properly
// allocated and in cache.
static void* my_calloc(size_t size1,size_t size2){
  size_t size=size1*size2;
  void* ret=malloc(size);

  if(ret==NULL){
    abort();
  }

  memset(ret,0,size);
  return ret;
}

int between(int start, int end){
  int len = end-start;
  int size = rand()%len;
  return start+size;
}



  /**********************/
 /* Disturbance threads */
/**********************/
int num_disturbance_threads_started = 0;
char *from_mem;
char *to_mem;
int mem_size;
bool run_disturbance=true;
pthread_t threads[200]={0};

static void *disturbance_thread(void *arg){
  int thread_num = atomic_add_get_oldval(&num_disturbance_threads_started,1);
  int cpu = (thread_num%3)+1; // cpu=1,2 or 3. (The test is run on cpu 0.)

  if(test_num==2){
    bound_thread_to_cpu(cpu);
    set_realtime(SCHED_FIFO,60);
    printf("Started disturbance thread on CPU %d\n",cpu);
  }

  if(test_num==1){
    int max_size = 8192*16;
    while(run_disturbance){
      int start = between(1,mem_size-max_size-10);
      int end   = between(start,start+max_size);
      memcpy(to_mem+start,from_mem+start,end-start);
    }
  }else{
```

```
    int mem_size=20*1014*1024;
    void *from_mem  = my_calloc(1,mem_size);
    void *to_mem    = my_calloc(1,mem_size);
    while(run_disturbance){
      memcpy(to_mem,from_mem,mem_size);
      usleep(1); // Must sleep a little bit to avoid crashing computer.
    }
  }

  printf("Disturbance thread %d finished\n",cpu);
  return NULL;
}

void start_disturbance_threads(int num){
  int i;

  mem_size  = 80*1024*1024;
  from_mem  = my_calloc(1,mem_size);
  to_mem    = my_calloc(1,mem_size);

  for(i=0;i<num;i++){
    pthread_create(&threads[i],NULL,disturbance_thread,NULL);
  }
  while(num_disturbance_threads_started<num)
    sleep(1);
}

void stop_disturbance_threads(int num){
  int i;
  run_disturbance=false;
  for(i=0;i<num;i++){
    pthread_cancel(threads[i]);
    pthread_join(threads[i],NULL);
  }
}



  /*****************************/
 /* Test snapshot performance. */
/*****************************/
float MB_per_ms(double ms){
  return (((double)heap_size) / ms) / (1024.0*1024.0);
}

static void *snapshot_thread(void *arg){
  int cpu = 0;
  void *from_mem  = my_calloc(1,heap_size);
  void *to_mem    = my_calloc(1,heap_size);

  bound_thread_to_cpu(cpu);

  if(from_mem==NULL || to_mem==NULL){
    fprintf(stderr,"Out of memory for size %d on cpu %d\n",heap_size,cpu);
    free(from_mem);
    free(to_mem);
    return NULL;
  }

  set_realtime(SCHED_FIFO,60); // jack thread priority
  usleep(20);
```

```
  double verystart=get_ms();
  double best_time = 0.0;
  double worst_time = 0.0;

  for(;;){
    double start=get_ms();
    memcpy(to_mem,from_mem,heap_size);
    double time=get_ms()-start;

    add_sample(time);

    if(best_time==0.0 || time<best_time)
      best_time=time;
    if(time>worst_time)
      worst_time=time;
    if(start-verystart > 1000)
      break;

    usleep(10); // This is: 1. Realistic
                //           2. Necessary to avoid stalling the computer.
  }

  fprintf(stderr,
          "size: %d, cpu: %d. Best: %f (%f MB/ms), Worst: %f (%f MB/ms)\n",
          heap_size,cpu,
          (float)best_time,(float) MB_per_ms(best_time),
          (float)worst_time,(float) MB_per_ms(worst_time));

  free(from_mem);
  free(to_mem);

  return NULL;
}

void perform_test(){
  pthread_t thread={0};

  pthread_create(&thread,NULL,snapshot_thread,NULL);
  pthread_join(thread,NULL);

  sleep(1);
}



  /*****************************/
 /* Run tests.                */
/*****************************/
int main(int argc, char **argv){
  int i;
  double min,q1,median,q3,max;

  FILE *file = NULL;

  test_num = atoi(argv[1]);

  if(test_num==0)
    file = fopen("/tmp/snapshot_cpu1.stats","w");
  else if(test_num==1){
    file = fopen("/tmp/snapshot_cpu4.stats","w");
    num_disturbance_threads = 40;
  }else if(test_num==2){
```

```
    file = fopen("/tmp/snapshot_cpu4rt.stats","w");
    num_disturbance_threads = 3;
  }else{
    file = fopen("/tmp/snapshot_extra.stats","w");
    num_disturbance_threads = 3;
    test_num=2;
  }

  fprintf(file,
          "# 1. size \t 2. min \t 4. q1 \t 6. "
          "median \t 8. q3 \t 10. max \n"
          "# (3,5,7,9 and 11 are MB/ms)\n");

  start_disturbance_threads(num_disturbance_threads);

  for(i=0;i<NUM_TESTS;i++){
    heap_size=(int)(busy_lengths[i] * 1024.0*1024.0);
    reset_statistics();
    perform_test();
    get_statistics(&min,&q1,&median,&q3,&max);
    fprintf(file,"%f\t %f %f\t %f %f\t %f %f\t %f %f\t %f %f\n",
            ((float)(heap_size))/(1024.0*1024.0),
            min,MB_per_ms(min),
            q1,MB_per_ms(q1),
            median,MB_per_ms(median),
            q3,MB_per_ms(q3),
            max,MB_per_ms(max));
  }

  stop_disturbance_threads(num_disturbance_threads);

  fclose(file);
  return 0;
}
```

A.5 A program for finding $p(M)$

```
#define NUM_CPUS 4
#define CPU_MHZ 2667

#define _GNU_SOURCE
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <sched.h>
#include <stdbool.h>


static size_t heap_size = 50 * 1024 * 1024;


  /**********************/
 /* Threads and timing */
/**********************/
#if defined(__i386__)
// Also copied from the JACK source
typedef unsigned long long cycles_t;
static inline cycles_t get_cycles (void){
  unsigned long long ret;
  __asm__ __volatile__("rdtsc" : "=A" (ret));
  return ret;
}
#endif

static double get_ms(void){
#if defined(__x86_64)
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  return ts.tv_sec * 1000.0 + ((double)ts.tv_nsec) / 1000000.0;
#else
  return ((double)get_cycles()) / (1000.0*(double)CPU_MHZ);
#endif
}

static void bound_thread_to_cpu(int cpu){
  cpu_set_t set;
  CPU_ZERO(&set);
  CPU_SET(cpu,&set);
  pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &set);
}

static int set_pthread_priority(pthread_t pthread,int policy,int priority){
  struct sched_param par={0};
  par.sched_priority=priority;

  if ((pthread_setschedparam(pthread, policy, &par)) != 0) {
    abort();
    return 0;
  }
  return 1;
}
```

```
static void set_realtime(int type, int priority){
  set_pthread_priority(pthread_self(),type,priority);
}



  /**********************/
 /* buzy threads.      */
/**********************/
// These threads prevents others from
// using the memory bus
static void *busy_thread(void *arg){
  int cpu = (int)arg;
  bound_thread_to_cpu(cpu);
  set_realtime(SCHED_FIFO,1);
  while(true){
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
    __asm__ __volatile__("nop");
  }
  return NULL;
}

void start_busy_threads(void){
  static bool started=false;
  int cpu;
  if(started==false){
    for(cpu=1;cpu<NUM_CPUS;cpu++){
      pthread_t *thread=calloc(sizeof(pthread_t),1);
      pthread_create(thread,NULL,busy_thread,(void*)cpu);
    }
    started=true;
  }
}

// clear memory manually to make sure the memory is properly
// allocated and in cache.
static void* my_calloc(size_t size1,size_t size2){
  size_t size=size1*size2;
  void* ret=malloc(size);

  if(ret==NULL){
    fprintf(stderr,"my_calloc: malloc returned NULL.");
    abort();
  }

  memset(ret,0,size);
  return ret;
}
```

```
  /**********************/
 /* Find p(M)          */
/**********************/
// Called (once) from gc_thread
static long double calculate_snapshot_time(void){
  double  snapshot_time = 0.0;
  void    *from         = my_calloc(1,heap_size);
  void    *to           = my_calloc(1,heap_size);

  bound_thread_to_cpu(0);
  usleep(1);

  int n;
  int i;

  for(n=0;n<2;n++){
    printf("Test %d/2\n",n+1);
    double verystart=get_ms();
    set_realtime(SCHED_OTHER,0);
    for(i=0;;i++){
      double start=get_ms();
      memcpy(to,from,heap_size);
      double time=get_ms()-start;
      if(snapshot_time==0.0 || time<snapshot_time)
        snapshot_time=time;
      double sofar = start-verystart;
      if(sofar > 40000)
        break;
      if(sofar > 10000 && sofar < 20000)
        set_realtime(SCHED_RR,2);
      else if(sofar > 20000 && sofar < 30000)
        set_realtime(SCHED_FIFO,10);
      else if(sofar > 30000)
        set_realtime(SCHED_FIFO,70);
      usleep((int)time);
    }
    start_busy_threads();
  }

  printf("snapshot time: %fms (%f MB/ms). Size: %d\n",
         (float)snapshot_time,
         (float) (heap_size / snapshot_time) / (1024.0*1024.0),
         heap_size
         );

  set_realtime(SCHED_OTHER,0);
  return ((long double)heap_size) / (((long double)snapshot_time) * 1024.0 * 1024.0);
}

int main(int argc, char **argv){
  printf("\n\n*****************************\n");
  printf("This program calculates snapshot performance:\n\n");
  printf("  1.   Make sure you are root. \n");
  printf("  2.   No other programs should run while running this program.\n");
  printf("  3.   Snapshot performance is stored in %s. (MB/ms)\n",argv[1]);
  printf("*****************************\n\n");
  long double result = calculate_snapshot_time();
  FILE *file = fopen(argv[1],"w");
  fprintf(file,"%Lf\n",result);
  fclose(file);
  return 0;
}
```

B Benchmark Data

B.1 Snapshot Benchmark Data

p(M) / Realtime kernel
5.450805 MB/ms

p(M) / Non-Realtime kernel
5.502967 MB/ms

p(m,t,0) and p(m,t,3)
Generated figures for the data are shown in the subsequent pages. Following the figures are two additional tables. These tables contain data which were outside the range of the figures.

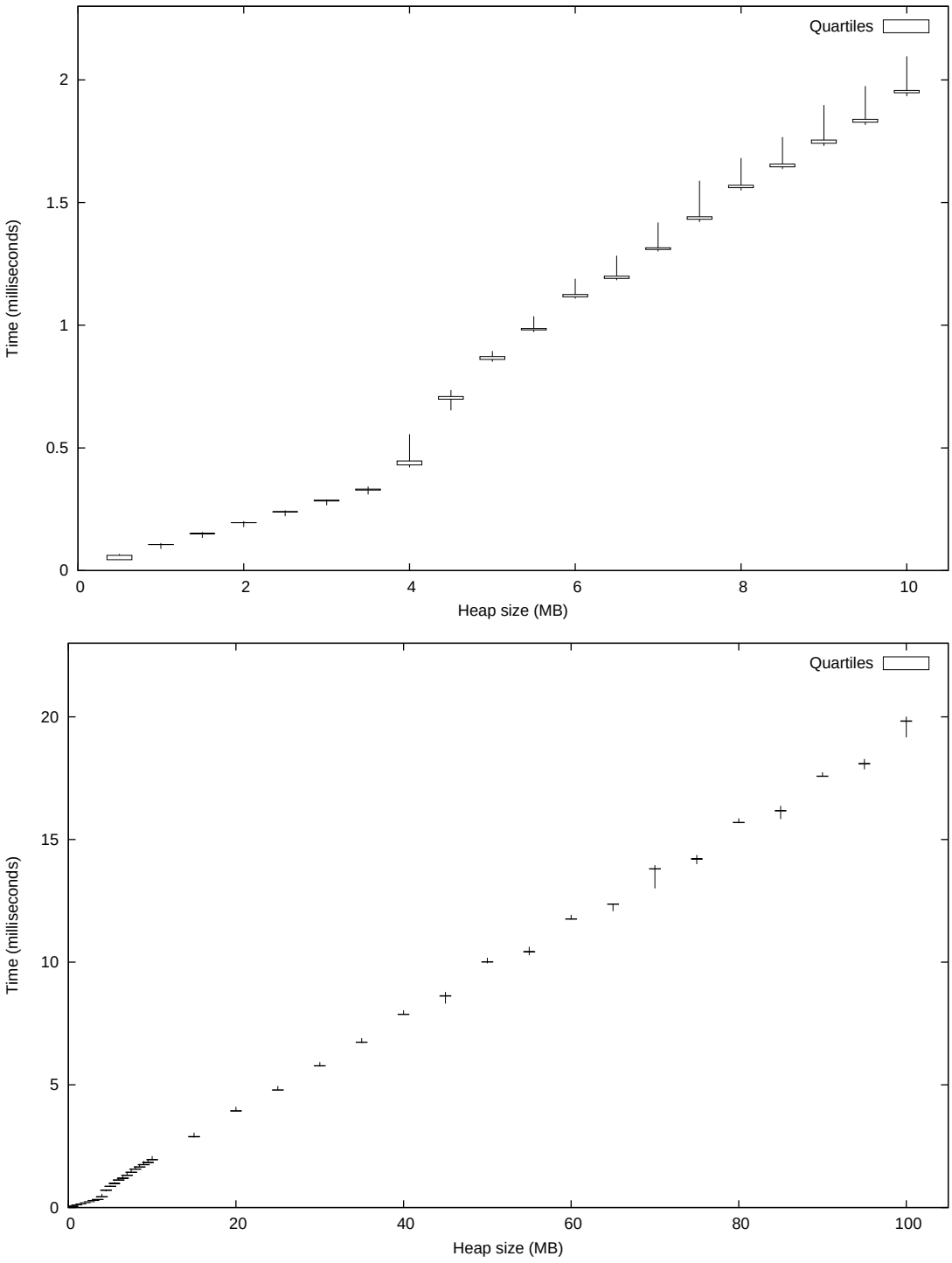

Figure B.1: $p(m, t, 0)$. Realtime kernel

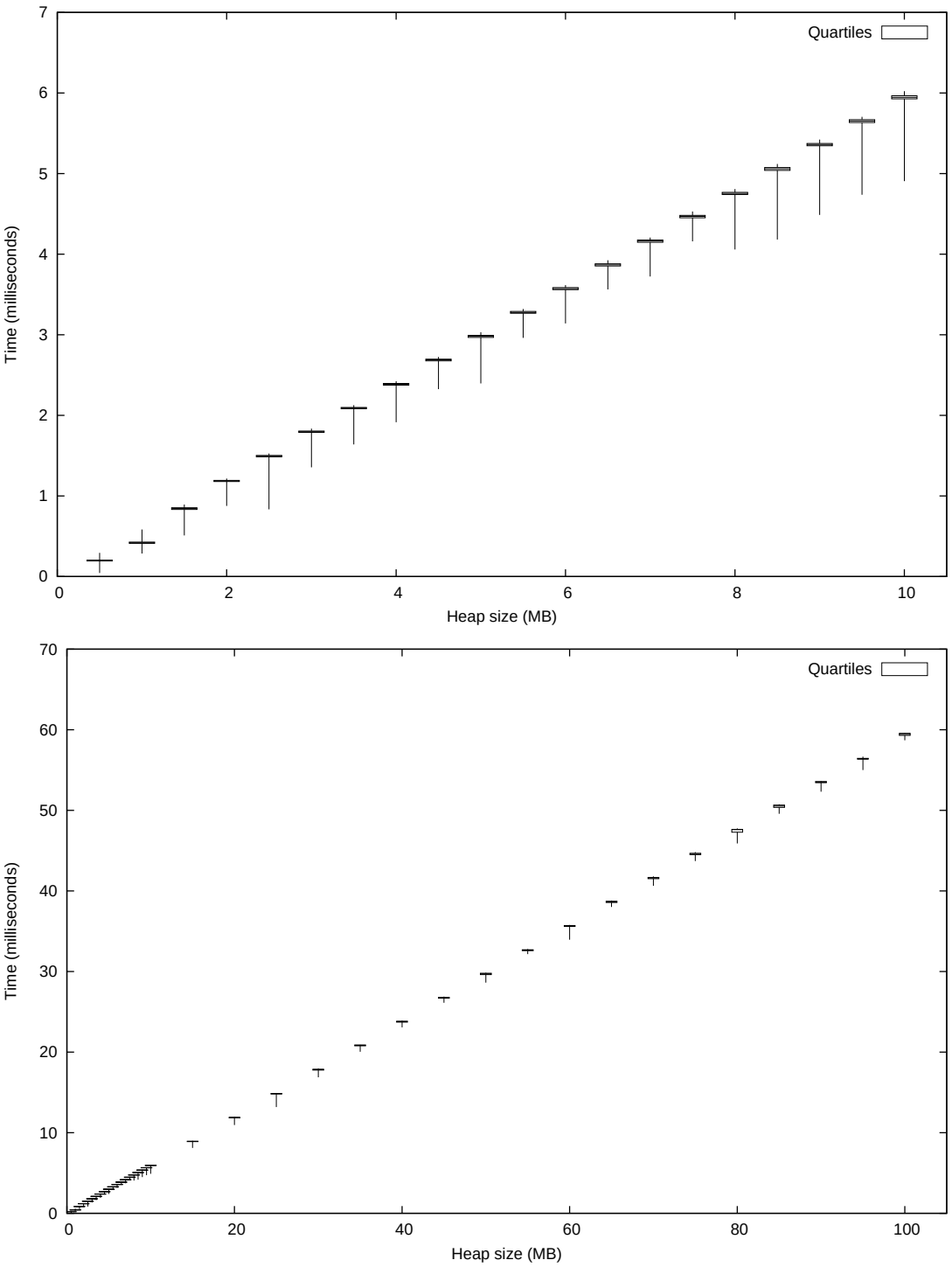

Figure B.2: $p(m, t, 3)$. Realtime kernel

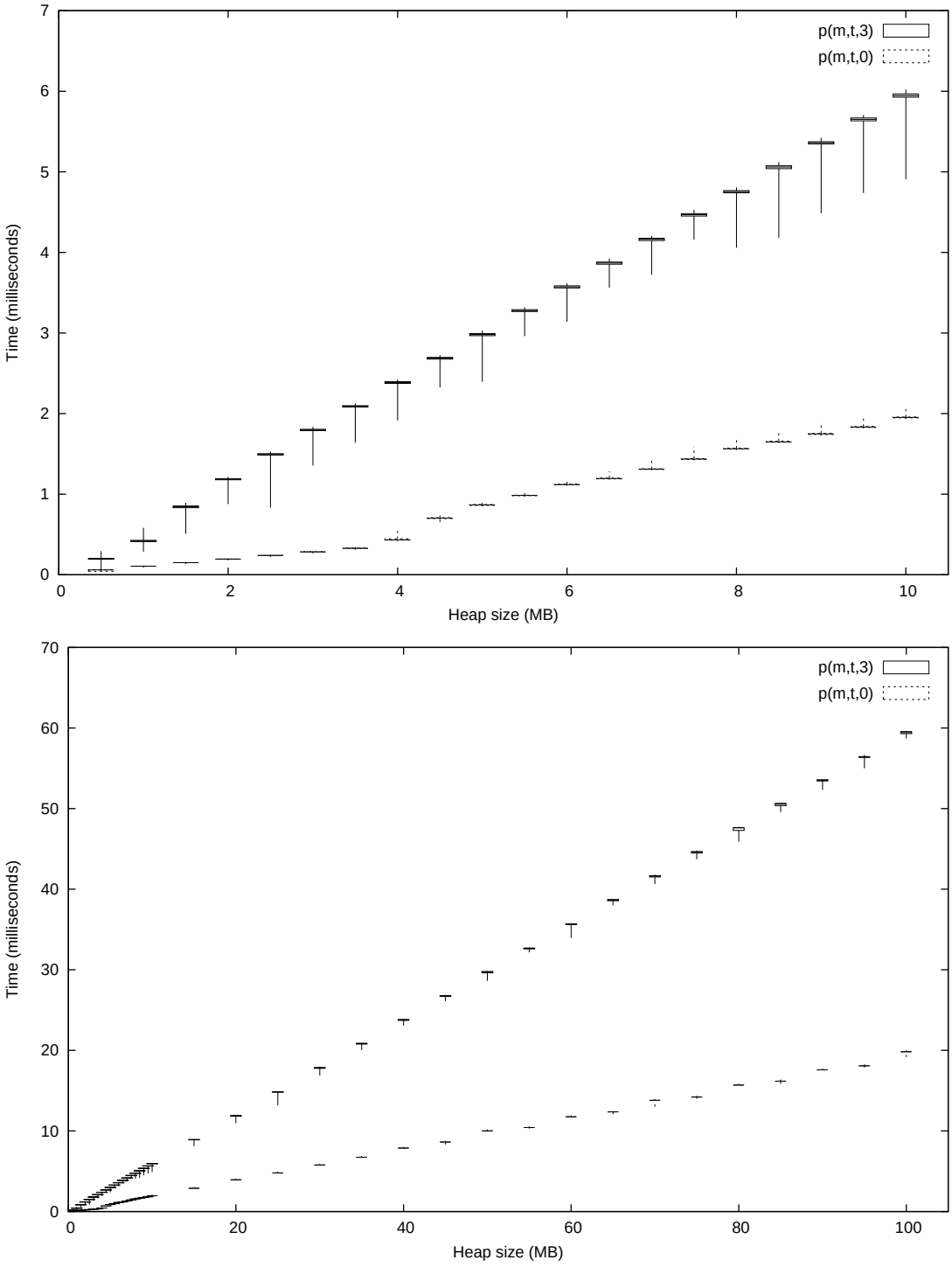

Figure B.3: $p(m, t, 0)$ and $p(m, t, 3)$. Realtime kernel

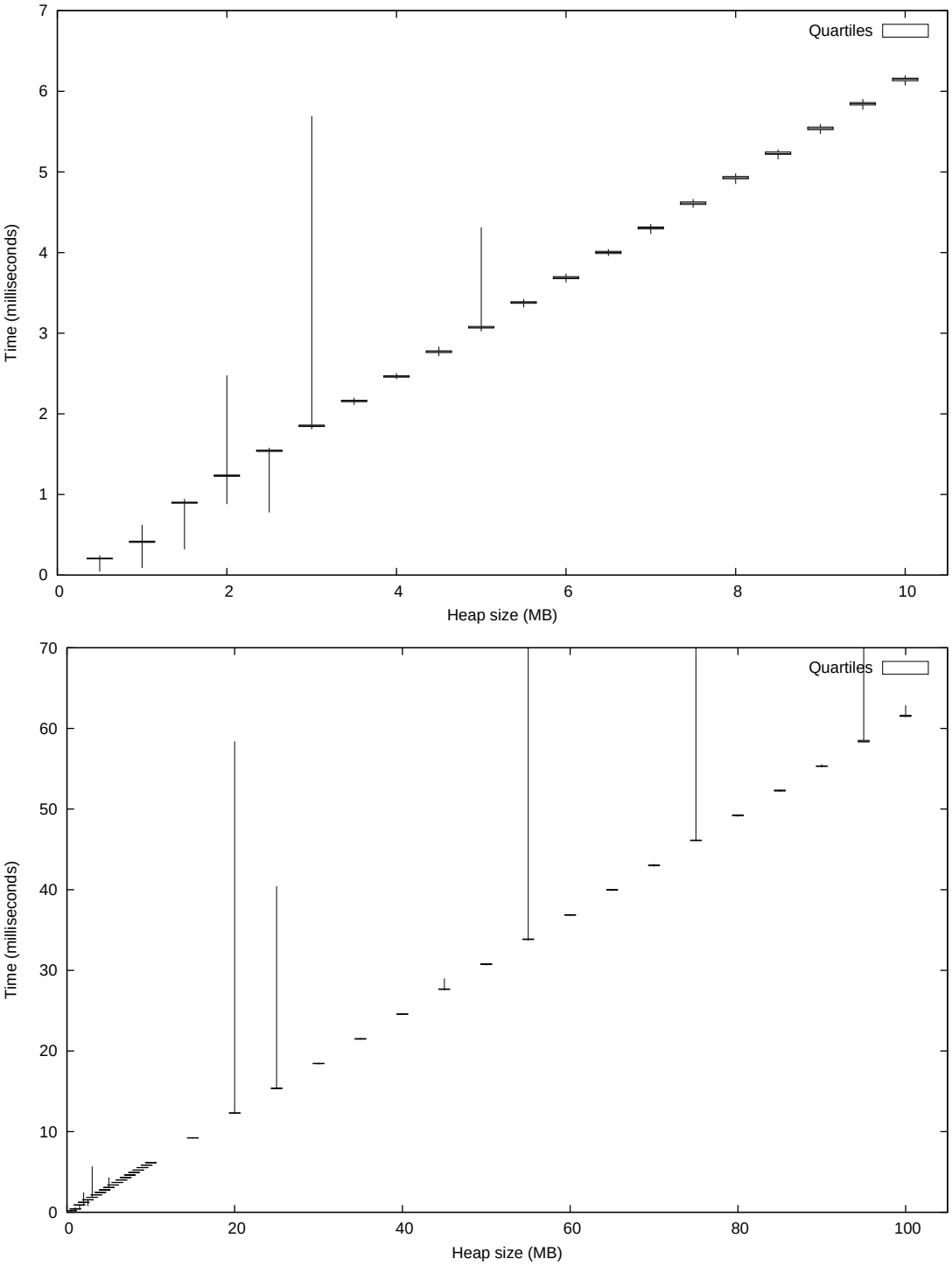

Figure B.4: $p(m,t,3)_{RT}$. Realtime kernel

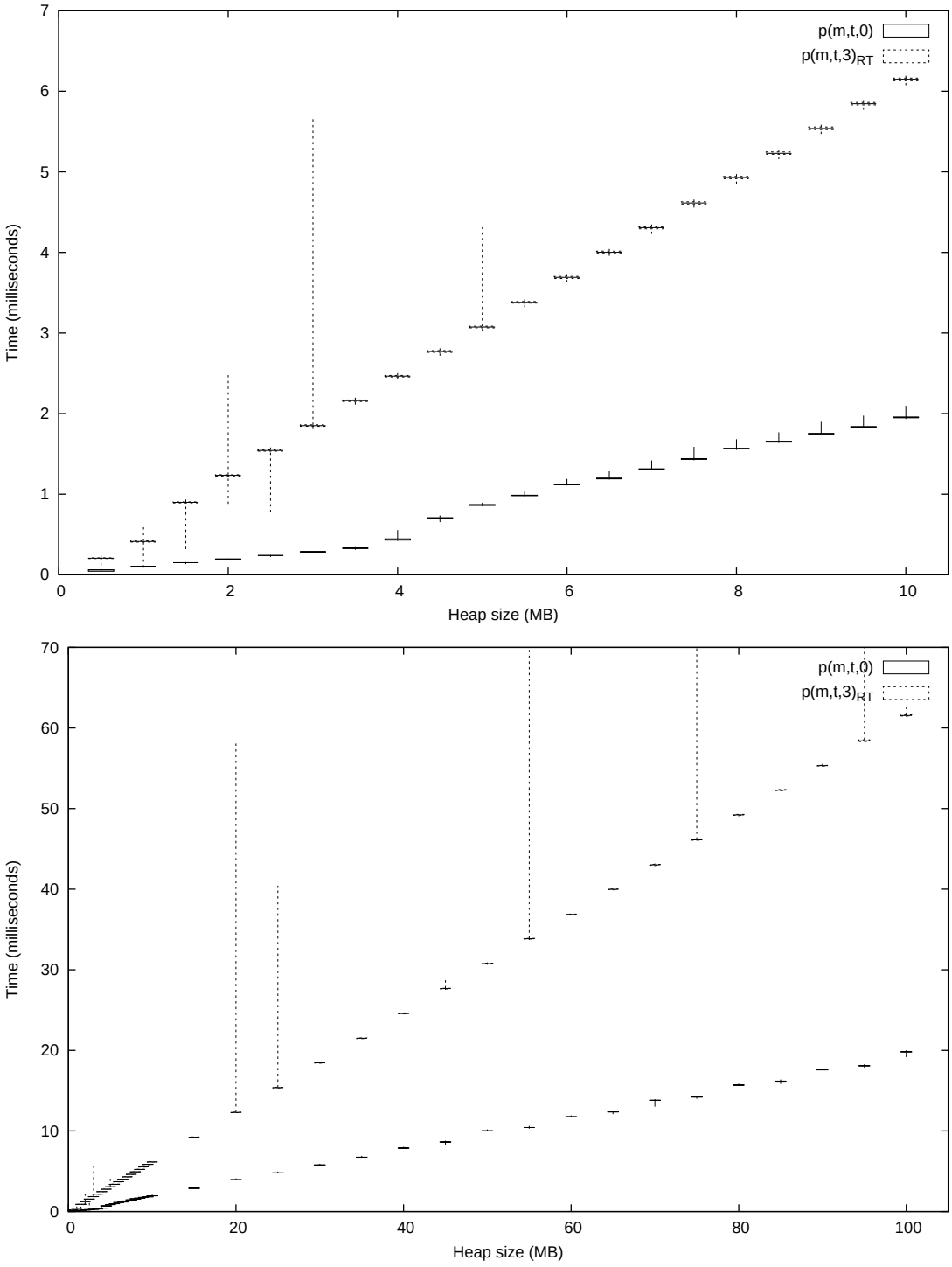

Figure B.5: $p(m,t,0)$ and $p(m,t,3)_{RT}$. Realtime kernel

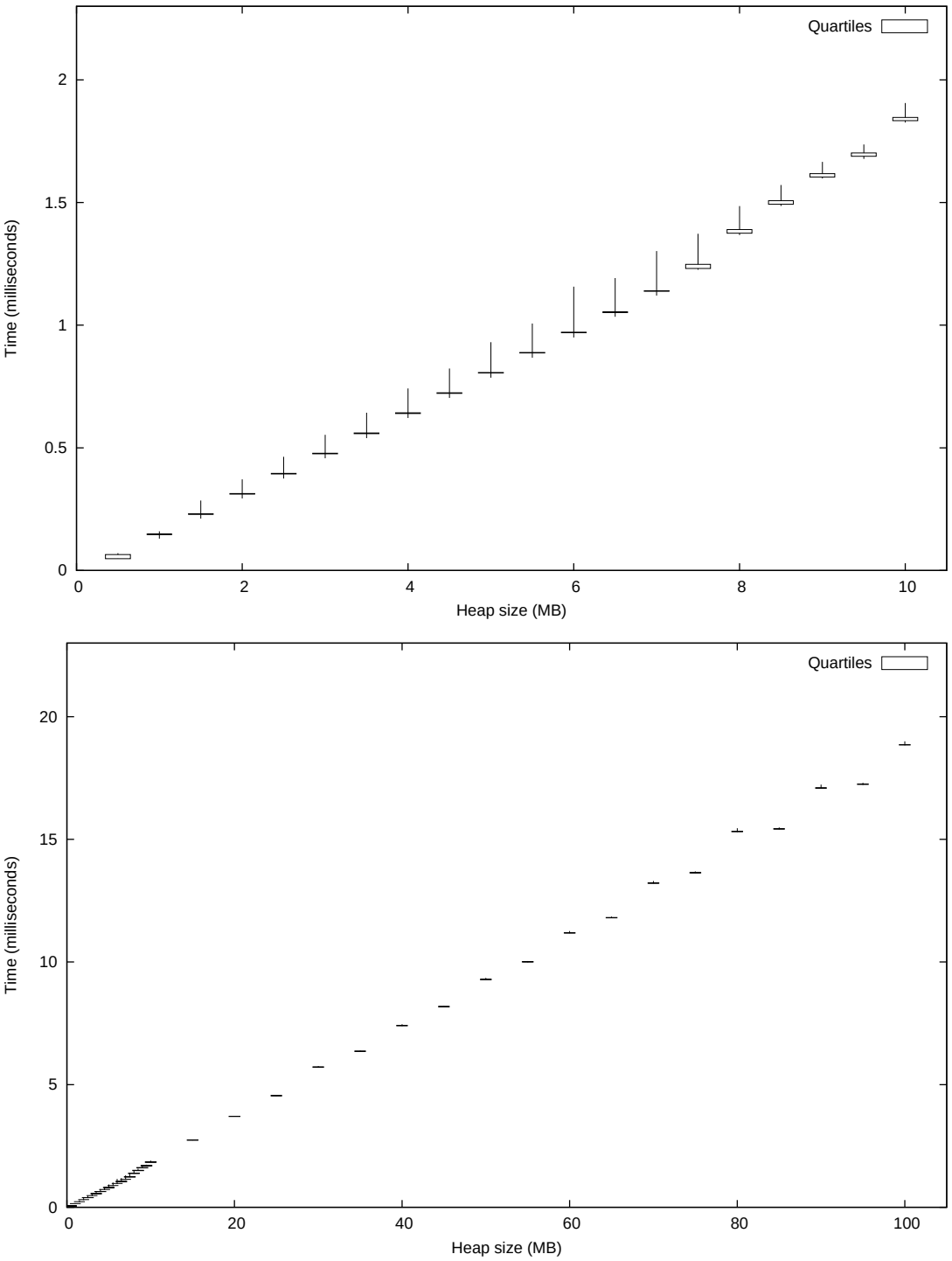

Figure B.6: $p(m, t, 0)$. Non-Realtime kernel

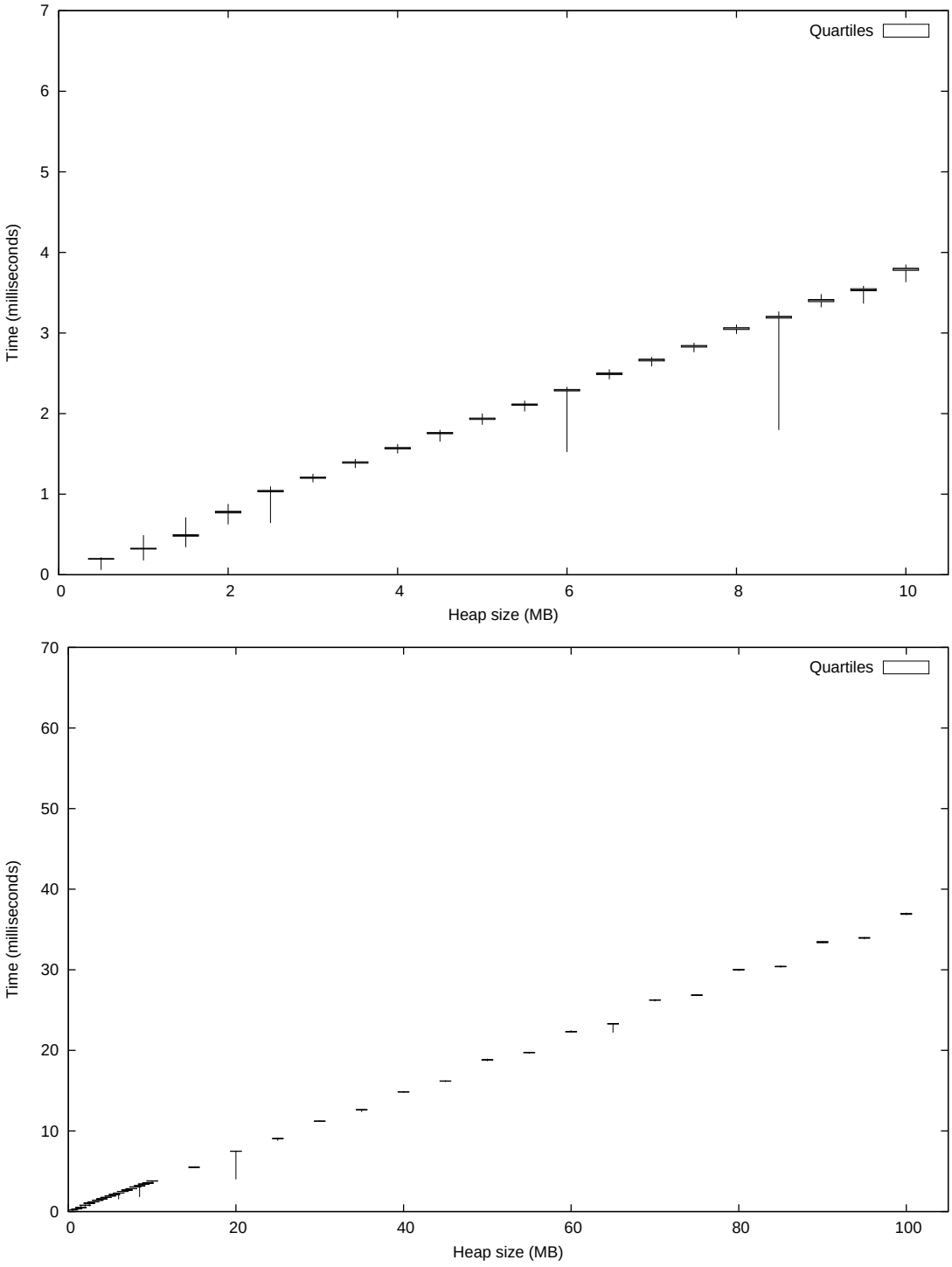

Figure B.7: $p(m, t, 3)$. Non-Realtime kernel

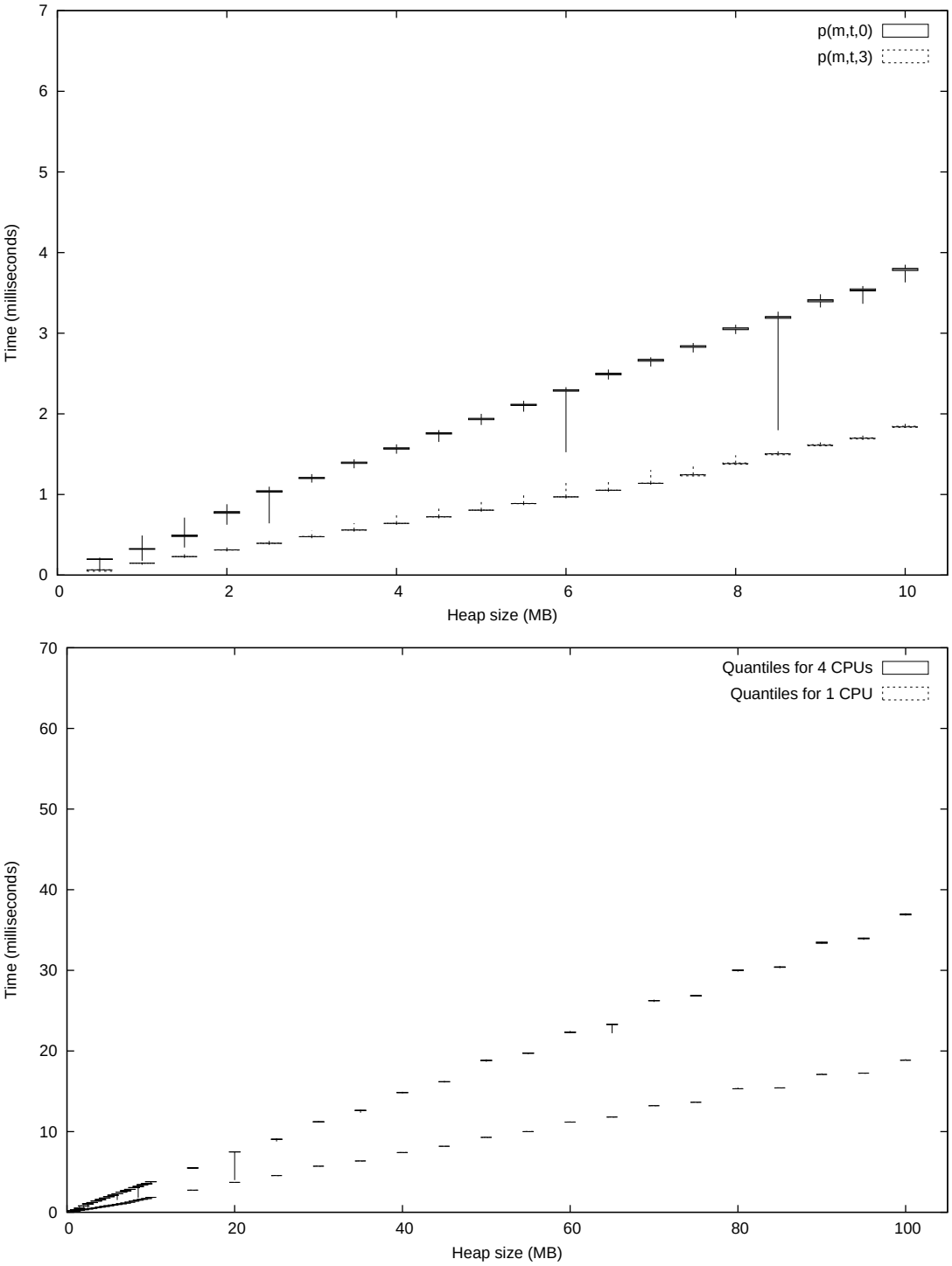

Figure B.8: $p(m, t, 0)$ and $p(m, t, 3)$. Non-Realtime kernel

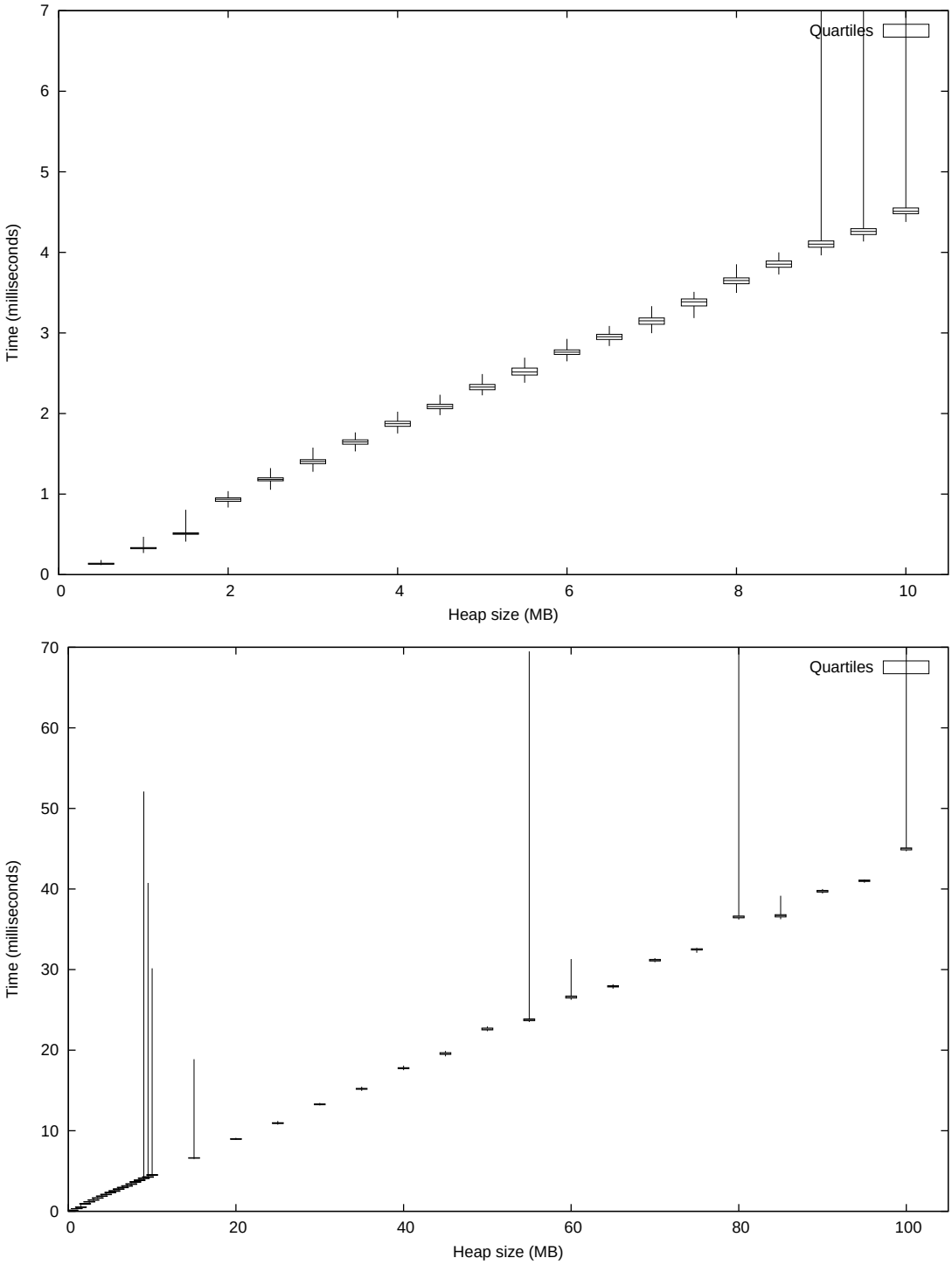

Figure B.9: $p(m,t,3)_{RT}$. Non-Realtime kernel

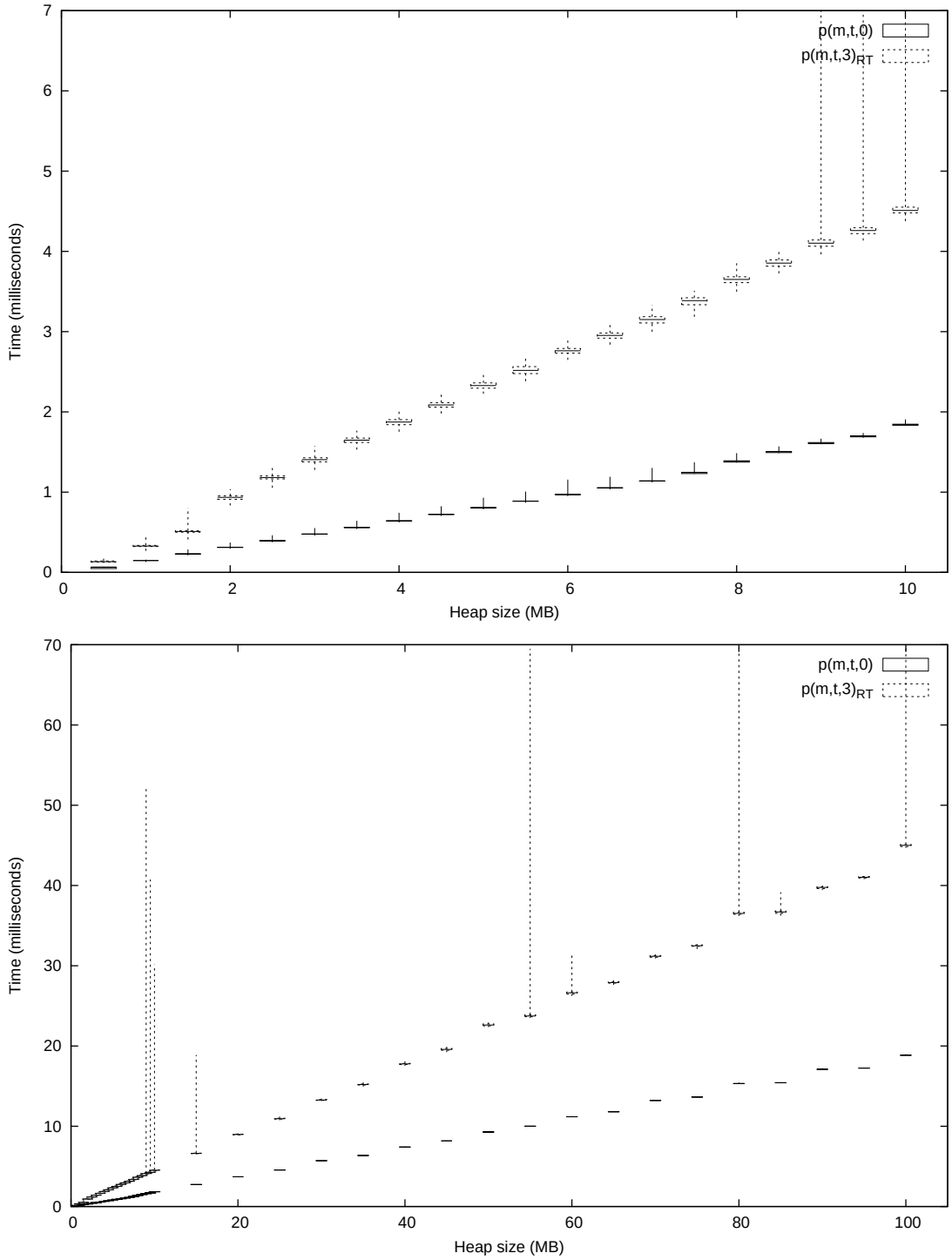

Figure B.10: $p(m,t,0)$ and $p(m,t,3)_{RT}$. Non-Realtime kernel

Heap Size	Best Case	Quartile 1	Median	Quartile 3	Worst Case
0.5	0.044684	0.201513	0.205722	0.21049	0.244741
1.0	0.090766	0.406673	0.413744	0.421105	0.62399
1.5	0.319624	0.892018	0.898487	0.905492	0.94508
2.0	0.88051	1.226379	1.233897	1.240819	2.477854
2.5	0.777489	1.534247	1.543069	1.551792	1.58024
3.0	1.808099	1.842202	1.851516	1.860313	5.693626
3.5	2.111952	2.150712	2.160019	2.168696	2.200864
4.0	2.429752	2.454577	2.464789	2.473356	2.505039
4.5	2.716735	2.760598	2.771154	2.781588	2.833629
5.0	3.023424	3.063772	3.074317	3.084277	4.312876
5.5	3.319325	3.370697	3.380964	3.390649	3.423952
6.0	3.629555	3.675982	3.690821	3.701101	3.740403
6.5	3.957912	3.98929	4.001467	4.016013	4.042112
7.0	4.22996	4.292955	4.306593	4.318083	4.353545
7.5	4.556502	4.596762	4.61305	4.628151	4.662857
8.0	4.851208	4.914616	4.929897	4.943724	4.982634
8.5	5.158074	5.218149	5.230617	5.246788	5.278299
9.0	5.470835	5.52381	5.537683	5.556325	5.593056
9.5	5.770604	5.830955	5.845609	5.860472	5.904267
10.0	6.073251	6.13068	6.14882	6.162698	6.196844
15.0	9.149275	9.201794	9.219796	9.24055	9.287625
20.0	12.215006	12.275261	12.307864	12.323943	58.405504
25.0	15.277174	15.345837	15.372462	15.392052	40.446793
30.0	18.331121	18.420441	18.455696	18.484811	18.552087
35.0	21.405816	21.469818	21.499729	21.533324	21.590359
40.0	24.453881	24.524445	24.567339	24.6013	24.700613
45.0	27.51315	27.631313	27.673245	27.701242	29.002815
50.0	30.625432	30.71071	30.767115	30.815633	30.911777
55.0	33.696726	33.81747	33.855549	33.895666	77.83564
60.0	36.74775	36.826682	36.866575	36.90428	36.966305
65.0	39.863466	39.948586	39.980855	40.009263	40.07049
70.0	42.85882	42.966028	43.020448	43.063043	43.184481
75.0	46.027232	46.084153	46.119159	46.156846	96.207538
80.0	49.045	49.171403	49.211934	49.268954	49.321261
85.0	52.16315	52.237321	52.286238	52.355175	52.392466
90.0	55.182439	55.27874	55.331759	55.362017	55.547447
95.0	58.293465	58.322847	58.411871	58.525462	71.417632
100.0	61.394761	61.487336	61.547231	61.632222	62.867169

Table B.1: $p(m, t, 3)_{RT}$. Realtime kernel

Heap Size	Best Case	Quartile 1	Median	Quartile 3	Worst Case
0.5	0.119428	0.12969	0.133911	0.141659	0.182272
1.0	0.268068	0.322256	0.327864	0.336085	0.470315
1.5	0.411181	0.501039	0.509233	0.518935	0.805754
2.0	0.83359	0.911063	0.935347	0.953201	1.036556
2.5	1.053996	1.164268	1.181555	1.202127	1.322605
3.0	1.276522	1.377848	1.404454	1.427671	1.576648
3.5	1.529345	1.619013	1.648051	1.67324	1.76612
4.0	1.753401	1.840711	1.873677	1.903075	2.023181
4.5	1.979419	2.059194	2.087064	2.115564	2.232987
5.0	2.225145	2.29575	2.327579	2.360214	2.489712
5.5	2.382325	2.477716	2.516738	2.56424	2.693243
6.0	2.64749	2.732299	2.760987	2.789467	2.926828
6.5	2.83771	2.919603	2.953132	2.983222	3.087096
7.0	2.996878	3.107895	3.149925	3.185415	3.332329
7.5	3.185212	3.335675	3.386033	3.421975	3.510083
8.0	3.497617	3.610984	3.649939	3.683468	3.852473
8.5	3.725207	3.815484	3.85415	3.892898	4.000653
9.0	3.962811	4.064238	4.102174	4.142302	52.099063
9.5	4.135174	4.221416	4.260215	4.293364	40.747177
10.0	4.378015	4.479449	4.510159	4.55127	30.151489
15.0	6.493906	6.588685	6.613612	6.648272	18.881081
20.0	8.857371	8.932548	8.982968	9.011471	9.124744
25.0	10.756475	10.891002	10.936092	11.000027	11.208493
30.0	13.152494	13.225758	13.273104	13.320987	13.451196
35.0	14.937004	15.142429	15.19983	15.248663	15.475206
40.0	17.519118	17.699605	17.763825	17.829237	18.087501
45.0	19.236299	19.478394	19.571739	19.684045	19.905904
50.0	22.3575	22.5081	22.616673	22.744249	22.940387
55.0	23.493413	23.649004	23.729114	23.884909	69.48738
60.0	26.251881	26.469786	26.629803	26.721885	31.308868
65.0	27.630792	27.839276	27.897934	27.998989	28.175497
70.0	30.905873	31.058561	31.179959	31.26528	31.419917
75.0	32.085084	32.426483	32.487429	32.5861	32.695523
80.0	36.205265	36.416786	36.516823	36.647598	83.829661
85.0	36.24527	36.52975	36.704472	36.81466	39.155691
90.0	39.438141	39.620114	39.759124	39.830983	40.005963
95.0	40.784065	40.930771	41.021931	41.106647	41.184951
100.0	44.68979	44.857	44.993319	45.098891	91.644479

Table B.2: $p(m, t, 3)_{RT}$. Non-Realtime kernel

B.2 Garbage Collection Benchmark Data

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	236160	1024	3936	4096	60
audio (ms)	10285.524414	0.388883	0.989849	3.168003	10391
alloc (ms)	8.119198	0.000009	0.000042	0.005393	193751
free (ms)	20.542446	0.000025	0.000084	0.003853	243381
sweep (ms)	25.273035	0.000123	0.002432	0.027715	10390
mark (ms)	339.094635	0.020556	0.065273	0.199261	5195
roots snapshot (ms)	9.528028	0.001452	0.001834	0.012814	5195
partial snapshot (ms)	22.562778	0.000798	0.004393	0.010203	5136
full snapshot (ms)	43.393547	0.719588	0.735484	0.746814	59

Table B.3: Rollendurch. Test 1

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	235888	1024	3931	4096	60
audio (ms)	10246.692383	0.386956	0.987728	3.150688	10374
alloc (ms)	7.626374	0.000009	0.000039	0.023655	193681
free (ms)	20.106043	0.000021	0.000083	0.003868	243235
sweep (ms)	30.126350	0.000124	0.002904	0.021929	10374
mark (ms)	394.431549	0.023856	0.076042	0.224829	5187
roots snapshot (ms)	10.912249	0.001573	0.002104	0.027582	5187
partial snapshot (ms)	23.870722	0.000853	0.004655	0.009941	5128
full snapshot (ms)	43.488136	0.733704	0.737087	0.747361	59

Table B.4: Rollendurch. Test 2

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	237392	1648	3956	4096	60
audio (ms)	10258.189453	0.389287	0.989123	3.166529	10371
alloc (ms)	7.803329	0.000009	0.000040	0.025569	193663
free (ms)	20.294376	0.000022	0.000083	0.005390	243194
sweep (ms)	30.426102	0.000141	0.002935	0.031786	10368
mark (ms)	372.616241	0.021354	0.071878	0.209482	5184
roots snapshot (ms)	10.316799	0.001506	0.001990	0.018997	5184
partial snapshot (ms)	24.075989	0.000840	0.004698	0.010431	5125
full snapshot (ms)	43.287762	0.716754	0.733691	0.740760	59

Table B.5: Rollendurch. Test 3

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	234752	1024	3912	4096	60
audio (ms)	10264.081055	0.387004	0.989309	3.153840	10375
alloc (ms)	8.278701	0.000010	0.000043	0.014266	193667
free (ms)	20.637423	0.000019	0.000085	0.004409	243225
sweep (ms)	30.093145	0.000121	0.002901	0.023183	10374
mark (ms)	358.098297	0.021017	0.069038	0.218958	5187
roots snapshot (ms)	9.307538	0.001450	0.001794	0.027978	5187
partial snapshot (ms)	23.157463	0.000810	0.004516	0.012048	5128
full snapshot (ms)	43.181957	0.712121	0.731898	0.741024	59

Table B.6: Rollendurch. Test 4

	Total	Minimum	Average	Maximum	#
snapshot length (kB)	235088	1024	3918	4096	60
audio (ms)	10267.342773	0.389080	0.989623	3.155974	10375
alloc (ms)	7.988904	0.000009	0.000041	0.033227	193693
free (ms)	20.622456	0.000022	0.000085	0.003561	243251
sweep (ms)	31.494452	0.000124	0.003036	0.030742	10374
mark (ms)	367.501099	0.021045	0.070850	0.201266	5187
roots snapshot (ms)	9.986195	0.001468	0.001925	0.016564	5187
partial snapshot (ms)	24.250828	0.000798	0.004729	0.010016	5128
full snapshot (ms)	43.427841	0.726436	0.736065	0.746623	59

Table B.7: Rollendurch. Test 5

	Total	Minimum	Average	Maximum	#
audio (ms)	10250.626953	0.388990	0.988203	3.158262	10373
alloc (ms)	8.411024	0.000010	0.000043	0.005000	193679
free (ms)	20.516733	0.000025	0.000084	0.004039	243229
sweep (ms)	24.981117	0.000097	0.002408	0.017209	10374
mark (ms)	350.102631	0.023651	0.067496	0.178047	5187

Table B.8: Mark-and-sweep. Test 1

	Total	Minimum	Average	Maximum	#
audio (ms)	10270.765625	0.385986	0.990049	3.167598	10374
alloc (ms)	8.677264	0.000009	0.000045	0.017737	193673
free (ms)	20.499599	0.000025	0.000084	0.003661	243228
sweep (ms)	28.433460	0.000099	0.002740	0.017347	10376
mark (ms)	376.455780	0.025758	0.072563	0.317276	5188

Table B.9: Mark-and-sweep. Test 2

	Total	Minimum	Average	Maximum	#
audio (ms)	10276.000977	0.384526	0.990553	3.166785	10374
alloc (ms)	8.346857	0.000009	0.000043	0.014776	193673
free (ms)	20.288712	0.000024	0.000083	0.003579	243219
sweep (ms)	30.474638	0.000114	0.002938	0.019921	10374
mark (ms)	366.550751	0.023624	0.070667	0.207243	5187

Table B.10: Mark-and-sweep. Test 3

	Total	Minimum	Average	Maximum	#
audio (ms)	10264.071289	0.390728	0.988832	3.162604	10380
alloc (ms)	8.186666	0.000009	0.000042	0.014115	193709
free (ms)	20.535469	0.000028	0.000084	0.003715	243291
sweep (ms)	29.888170	0.000111	0.002879	0.019555	10382
mark (ms)	367.544586	0.023667	0.070804	0.314230	5191

Table B.11: Mark-and-sweep. Test 4

	Total	Minimum	Average	Maximum	#
audio (ms)	10263.000000	0.388877	0.989300	3.162440	10374
alloc (ms)	8.146610	0.000009	0.000042	0.022571	193703
free (ms)	20.668045	0.000025	0.000085	0.003882	243257
sweep (ms)	31.732052	0.000111	0.003059	0.021273	10374
mark (ms)	369.281708	0.023198	0.071194	0.186257	5187

Table B.12: Mark-and-sweep. Test 5

Bibliography

[AEL88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Not.*, 23(7):11–20, 1988.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[Bak92] Henry G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.

[Bak94] Henry G. Baker. Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.*, 29(9):38–43, 1994.

[BCR03] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM.

[Ble10] Tim Blechmann. Supernova, a multiprocessor-aware synthesis server for SuperCollider. *Proceedings of the Linux Audio Conference 2010*, pages 141–146, 2010.

[BM03] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. *SIGPLAN Not.*, 38(11):344–358, 2003.

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

[Boe02] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. *SIGPLAN Not.*, 37(1):93–100, 2002.

[BR01] David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 207–235, London, UK, 2001. Springer-Verlag.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262, New York, NY, USA, 1984. ACM.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.

[Con63] Melvin E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[CR91] William Clinger and Jonathan Rees. Revised Report (4) On The Algorithmic Language Scheme, 1991.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[DN66] O.-J. Dahl and K. Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[FOL02] Dominique Fober, Yann Orlarey, and Stephane Letz. Lock-Free Techniques for Concurrent Access to Shared Objects. *Actes des Journes dInformatique Musicale JIM2002, Marseille*, pages 143-150, 2002.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, 1969.

[Gra06] Martin Grabmüller. Implementing Closures using Run-time Code Generation. *Technische Universität Berlin, research report*, 2006.

[Har93] M. G. Harbour. Real-time POSIX: an Overview. In *VVConex 93 International Conference, Moscu, June 1993.*, 1993.

[Hen94] R. Henriksson. Scheduling Real Time Garbage Collection. In *Proceedings of NWPER'94, Nordic Workshop on Programming Environment Research*, pages 253–266, 1994.

[HM01] Richard L. Hudson and J. Eliot B. Moss. Sapphire: copying GC without stopping the world. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 48–57, New York, NY, USA, 2001. ACM.

[JS83] David A. Jaffe and Julius O. Smith. Extensions of the Karplus-Strong Plucked-String Algorithm. *Computer Music Journal*, 7(2):56–69, 1983.

[JW99] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? *SIGPLAN Not.*, 34(3):26–36, 1999.

[Kac10] John Kacur. Real-Time Kernel For Audio and Visual Applications. *Proceedings of the Linux Audio Conference 2010*, pages 57–64, 2010.

[LFO05] S. Letz, D. Fober, and Y. Orlarey. jackdmp: Jack server for multi-processor machines. *Proceedings of the Linux Audio Conference 2010*, pages 29–37, 2005.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[LP06] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[Mat10] Kjetil Matheussen. Implementing a Polyphonic MIDI Software Synthesizer using Coroutines, Realtime Garbage Collection, Closures, Auto-Allocated Variables, Dynamic Scoping, and Continuation Passing Style Programming. *Proceedings of the Linux Audio Conference 2010*, pages 7–15, 2010.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.

[Min63] Marvin Minsky. A LISP Garbage Collector Algorithm Using Serial Secondary Storage. Technical report, Cambridge, MA, USA, 1963.

[MRCR04] 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*, pages 79–86, Washington, DC, USA, 2004. IEEE Computer Society.

[MRR+08] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. J. Wellings. Implementation of a constant-time dynamic storage allocator. *Softw. Pract. Exper.*, 38(10):995–1026, 2008.

[Nie77] Norman R. Nielsen. Dynamic memory allocation in computer simulation. *Commun. ACM*, 20(11):864–873, 1977.

[NO93] Scott Nettles and James O'Toole. Real-time replication garbage collection. *SIGPLAN Not.*, 28(6):217–226, 1993.

[ON94] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 34–42, New York, NY, USA, 1994. ACM.

[PFPS07] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[Pir98] Pekka P. Pirinen. Barrier techniques for incremental tracing. In *ISMM '98: Proceedings of the 1st international symposium on*

Memory management, pages 20–25, New York, NY, USA, 1998. ACM.

[POS90] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). 1990.

[PPS08] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 33–44, New York, NY, USA, 2008. ACM.

[RRR97] Gustavo Rodriguez-Rivera and Vincent F. Russo. Nonintrusive cloning garbage collection with stock operating system support. *Softw. Pract. Exper.*, 27(8):885–904, 1997.

[RWRF09] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for C. In *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, pages 39–48, New York, NY, USA, 2009. ACM.

[Sch94] W. Schottstaedt. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):30–37, 1994.

[Sie10] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *ISMM '10: Proceedings of the 2010 international symposium on Memory management*, pages 11–20, New York, NY, USA, 2010. ACM.

[Sis] Jeffrey Mark Siskind. Stalin - a STAtic Language Implementatio N, `http://cobweb.ecn.purdue.edu/~qobi/software.html`.

[SOLF03] Nicolas Scaringella, Yann Orlarey, Stephane Letz, and Dominique Fober. Automatic vectorization in Faust. *Actes des Journes d'Informatique Musicale JIM2003, Montbeliard - JIM*, 2003.

[Ste75] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[TS08] Tian Tian and Chiu-Pi Shih. Software Techniques for Shared-Cache Multi-Core Systems, *Intel Knowledge Base*, `http://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems/`, 2008.

[Uni97] *The Single UNIX Specification, Version 2.* The Open Group, Woburn, MA., 1997.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *IWMM '92: Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Real-Time Non-Copying Garbage Collection. *ACM OOPSLA Workshop on Memory Management and Garbage Collection*, 1993.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

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.

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

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

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.

Implementing a Polyphonic MIDI Software Synthesizer using Coroutines, Realtime Garbage Collection, Closures, Auto-Allocated Variables, Dynamic Scoping, and Continuation Passing Style Programming

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
k.s.matheussen@notam02.no

Abstract

This paper demonstrates a few programming techniques for low-latency sample-by-sample audio programming. Some of them may not have been used for this purpose before. The demonstrated techniques are: Realtime memory allocation, realtime garbage collector, storing instrument data implicitly in closures, auto-allocated variables, handling signal buses using dynamic scoping, and continuation passing style programming.

Keywords

Audio programming, realtime garbage collection, coroutines, dynamic scoping, Continuation Passing Style.

1 Introduction

This paper demonstrates how to implement a MIDI software synthesizer (MIDI soft synth) using some unusual audio programming techniques. The examples are written for *Snd-RT* [Matheussen, 2008], an experimental audio programming system supporting these techniques. The techniques firstly emphasize convenience (i.e. few lines of code, and easy to read and modify), and not performance. Snd-RT[1] runs on top of Snd[2] which again runs on top of the Scheme interpreter Guile.[3] Guile helps gluing all parts together.

It is common in music programming only to compute the sounds themselves in a realtime priority thread. Scheduling new notes, allocation of data, data initialization, etc. are usually performed in a thread which has a lower priority than the audio thread. Doing it this way helps to ensure constant and predictable CPU usage for the audio thread. But writing code that way is also more complicated. At least, when all samples are calculated one by one. If however the programming only concerns handling blocks of samples where we only control a signal graph, there are several high level alternatives which makes it relatively easy to do a straightforward implementation of a MIDI soft synth. Examples of such high level music programming systems are SuperCollider [McCartney, 2002], Pd [Puckette, 2002], CSound[4] and many others.

But this paper does not describe use of block processing. In this paper, all samples are individually calculated. The paper also explores possible advantages of doing everything, allocation, initialization, scheduling, etc., from inside the realtime audio thread.

At least it looks like everything is performed inside the realtime audio thread. The underlying implementation is free to reorganize the code any way it wants, although such reorganizing is not performed in Snd-RT yet.

Future work is making code using these techniques perform equally, or perhaps even better, than code where allocation and initialization of data is explicitly written not to run in the realtime audio thread.

2 MIDI software synthesizer

The reason for demonstrating a MIDI soft synth instead of other types of music programs such as a granular synthesis generator or a reverb, is that the behavior of a MIDI soft synth is well known, plus that a MIDI soft synth contains many common challenges in audio and music programming:

1. Generating samples. To hear sound, we need to generate samples at the *Audio Rate*.
2. Handling Events. MIDI data are read at a rate lower than the audio rate. This rate is commonly called the *Control Rate*.

[1] http://archive.notam02.no/arkiv/doc/snd-rt/
[2] http://ccrma.stanford.edu/software/snd/
[3] http://www.gnu.org/software/guile/guile.html
[4] http://www.csound.com

3. Variable polyphony. Sometimes no notes are playing, sometimes maybe 30 notes are playing.

4. Data allocation. Each playing note requires some data to keep track of frequency, phase, envelope position, volume, etc. The challenges are; How do we allocate memory for the data? When do we allocate memory for the data? How do we store the memory holding the data? When do we initialize the data?

5. Bus routing. The sound coming from the tone generators is commonly routed both through an envelope and a reverb. In addition, the tones may be autopanned, i.e. panned differently between two loudspeakers depending on the note height (similar to the direction of the sound coming from a piano or a pipe organ).

3 Common Syntax for the Examples

The examples are written for a variant of the programming language Scheme [Steele and Sussman, 1978]. Scheme is a functional language with imperative operators and static scoping.

A number of additional macros and special operators have been added to the language, and some of them are documented here because of the examples later in the paper.

(<rt-stalin>...) is a macro which first transforms the code inside the block into clean R4RS code [Clinger and Rees, 1991] understood by the Stalin Scheme compiler.[5] (Stalin Scheme is an R4RS compiler). After Stalin is finished compiling the code, the produced object file is dynamically linked into Snd-RT and scheduled to immediately run inside the realtime audio thread.

(define-stalin *signature* **...)** defines variables and functions which are automatically inserted into the generated Stalin scheme code if needed. The syntax is similar to *define*.

(spawn ...**)** spawns a new coroutine [Conway, 1963; Dahl and Nygaard, 1966]. Coroutines are stored in a priority queue and it is not necessary to explicitly call the spawned coroutine to make it run. The spawned coroutine will run automatically as soon[6] as the current coroutine yields (by calling ***yield*** or ***wait***), or the current coroutine ends.

Coroutines are convenient in music programming since it often turns out practical to let one dedicated coroutine handle only one voice, instead of mixing the voices manually. Furthermore, arbitrarily placed pauses and breaks are relatively easy to implement when using coroutines, and therefore, supporting dynamic control rate similar to ChucK [Wang and Cook, 2003] comes for free.

(wait n**)** waits n number of frames before continuing the execution of the current coroutine.

(sound ...**)** spawns a special kind of coroutine where the code inside ***sound*** is called one time per sample. (***sound*** coroutines are stored in a tree and not in a priority queue since the order of execution for ***sound*** coroutines depends on the bus system and not when they are scheduled to wake up.)

A simple version of the ***sound*** macro, called *my-sound*, can be implemented like this:

```
(define-stalin-macro (my-sound . body)
  `(spawn
     (while #t
       ,@body
       (wait 1))))
```

However, *my-sound* is inefficient compared to ***sound*** since *my-sound* is likely to do a coroutine context switch at every call to ***wait***.[7] ***sound*** doesn't suffer from this problem since it is run in a special mode. This mode makes it possible to run tight loops which does not cause a context switch until the next scheduled event.

(out <*channel***>** *sample***)** sends out data to the *current bus* at the *current time*. (the *current bus* and the *current time* can be thought of as global variables which are implicitly read from and written to by many

[5] Stalin - a STAtic Language ImplementatioN, http://cobweb.ecn.purdue.edu/ qobi/software.html.

[6] Unless other coroutines are placed earlier in the queue.

[7] I.e. if two or more *my-sound* blocks or ***sound*** blocks run simultaneously, and at least one of them is a *my-sound* block, there will be at least two coroutine context switches at every sound frame.

operators in the system)[8] By default, the *current bus* is connected to the sound card, but this can be overridden by using the ***in*** macro which is explained in more detail later.

If the *channel* argument is omitted, the *sample* is written both to channel 0 and 1.

It makes sense only to use ***out*** inside a ***sound*** block. The following example plays a 400Hz sine sound to the sound card:

```
(<rt-stalin>
  (let ((phase 0.0))
    (sound
      (out (sin phase))
      (inc! phase (hz->radians 400)))))
```

(range *varname start end ...***)** is a simple loop iterator macro which can be implemented like this:[9]

```
(define-macro (range varname start end . body)
  (define loop (gensym))
  `(let ,loop ((,varname ,start))
     (cond ((<,var ,end)
            ,@body
            (,loop (+ ,varname 1))))))
```

(wait-midi :*options ...***)** waits until MIDI data is received, either from an external interface, or from another program.

wait-midi has a few options to specify the kind of MIDI data it is waiting for. In the examples in this paper, the following options for ***wait-midi*** are used:

:***command note-on***
: Only wait for a *note on* MIDI message.

:***command note-off***
: Only wait for a *note off* MIDI message.

:***note*** *number*
: Only wait for a note which has MIDI note number *number*.

Inside the ***wait-midi*** block we also have access to data created from the incoming midi event. In this paper we use ***(midi-vol)*** for getting the velocity (converted to a number between 0.0 and 1.0), and ***(midi-note)*** for getting the MIDI note number.

[8]Internally, the *current bus* is a coroutine-local variable, while the *current time* is a global variable.

[9]The actual implementation used in *Snd-RT* also makes sure that "end" is always evaluated only one time.

:where is just another way to declare local variables. For example,

```
(+ 2 b
  :where b 50)
```

is another way of writing

```
(let ((b 50))
  (+ 2 b))
```

There are three reason for using ***:where*** instead of ***let***. The first reason is that the use of ***:where*** requires less parenthesis. The second reason is that reading the code sometimes sounds more natural this way. (I.e "add 2 and b, where b is 50" instead of "let b be 50, add 2 and b".) The third reason is that it's sometimes easier to understand the code if you know what you want to do with a variable, before it is defined.

4 Basic MIDI Soft Synth

We start by showing what is probably the simplest way to implement a MIDI soft synth:

```
(range note-num 0 128
  (<rt-stalin>
    (define phase 0.0)
    (define volume 0.0)
    (sound
      (out (* volume (sin phase))))
      (inc! phase (midi->radians note-num)))
    (while #t
      (wait-midi :command note-on :note note-num
        (set! volume (midi-vol)))
      (wait-midi :command note-off :note note-num
        (set! volume 0.0)))))
```

This program runs 128 instruments simultaneously. Each instrument is responsible for playing one tone. 128 variables holding volume are also used for communicating between the parts of the code which plays sound (running at the *audio rate*), and the parts of the code which reads MIDI information (running at the *control rate*[10]).

There are several things in this version which are not optimal. Most important is that you

[10]Note that the *control rate* in Snd-RT is *dynamic*, similar to the music programming system *ChucK*. *Dynamic control rate* means that the smallest available time-difference between events is not set to a fixed number, but can vary. In ChucK, control rate events are measured in floating numbers, while in Snd-RT the measurement is in frames. So In Chuck, the time difference can be very small, while in Snd-RT, it can not be smaller than 1 frame.

would normally not let all instruments play all the time, causing unnecessary CPU usage. You would also normally limit the polyphony to a fixed number, for instance 32 or 64 simultaneously sounds, and then immediately schedule new notes to a free instrument, if there is one.

5 Realtime Memory Allocation

As mentioned, everything inside ***<rt-stalin>*** runs in the audio realtime thread. Allocating memory inside the audio thread using the OS allocation function may cause surprising glitches in sound since it is not guaranteed to be an $O(1)$ allocator, meaning that it may not always spend the same amount of time. Therefore, *Snd-RT* allocates memory using the Rollendurchmesserzeitsammler [Matheussen, 2009] garbage collector instead. The memory allocator in Rollendurchmesserzeitsammler is not only running in $O(1)$, but it also allocates memory extremely efficiently. [Matheussen, 2009]

In the following example, it's clearer that instrument data are actually stored in closures which are allocated during runtime.[11] In addition, the 128 spawned coroutines themselves require some memory which also needs to be allocated:

```
(<rt-stalin>
  (range note-num 0 128
   (spawn
     (define phase 0.0)
     (define volume 0.0)
     (sound
       (out (* volume (sin phase))))
       (inc! phase (midi->radians note-num)))
     (while #t
       (wait-midi :command note-on :note note-num
         (set! volume (midi-vol)))
       (wait-midi :command note-off :note note-num
         (set! volume 0.0)))))
```

6 Realtime Garbage Collection. (Creating new instruments only when needed)

The previous version of the MIDI soft synth did allocate some memory. However, since all memory required for the lifetime of the program were allocated during startup, it was not necessary to free any memory during runtime.

But in the following example, we simplify the code further by creating new tones only when they are needed. And to do that, it is necessary to free memory used by sounds not playing anymore to avoid running out of memory. Luckily though, freeing memory is taken care of automatically by the Rollendurchmesserzeitsammler garbage collector, so we don't have to do anything special:

```
 1| (define-stalin (softsynth)
 2|   (while #t
 3|     (wait-midi :command note-on
 4|       (define osc  (make-oscil :freq (midi->hz (midi-note))))
 5|       (define tone (sound (out (* (midi-vol) (oscil osc)))))
 6|       (spawn
 7|          (wait-midi :command note-off :note (midi-note)
 8|            (stop tone))))))
 9|
10| (<rt-stalin>
11|   (softsynth))
```

In this program, when a *note-on* message is received at line 3, two coroutines are scheduled:

1. A ***sound*** coroutine at line 5.
2. A regular coroutine at line 6.

Afterwards, the execution immediately jumps back to line 3 again, ready to schedule new notes.

So the MIDI soft synth is still polyphonic, and contrary to the earlier versions, the CPU is now the only factor limiting the number of simultaneously playing sounds.[12]

7 Auto-Allocated Variables

In the following modification, the CLM [Schottstaedt, 1994] oscillator *oscil* will be implicitly and automatically allocated first time the function *oscil* is called. After the generator is allocated, a pointer to it is stored in a special memory slot in the current coroutine.

Since *oscil* is called from inside a ***sound*** coroutine, it is natural to store the generator in the coroutine itself to avoid all tones using the same oscillator, which would happen if the auto-allocated variable had been stored in a global variable. The new definition of *softsynth* now looks like this:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))
```

[11] Note that memory allocation performed before any *sound* block can easily be run in a non-realtime thread before scheduling the rest of the code to run in realtime. But that is just an optimization.

[12] Letting the CPU be the only factor to limit polyphony is not necessarily a good thing, but doing so in this case makes the example very simple.

The difference between this version and the previous one is subtle. But if we instead look at the reverb instrument in the next section, it would span twice as many lines of code, and the code using the reverb would require additional logic to create the instrument.

8 Adding Reverb. (Introducing signal buses)

A MIDI soft synth might sound unprofessional or unnatural without some reverb. In this example we implement John Chowning's reverb[13] and connect it to the output of the MIDI soft synth by using the built-in signal bus system:

```
(define-stalin (reverb input)
  (delay :size (* .013 (mus-srate))
    (+ (comb :scaler 0.742 :size  9601 allpass-composed)
       (comb :scaler 0.733 :size 10007 allpass-composed)
       (comb :scaler 0.715 :size 10799 allpass-composed)
       (comb :scaler 0.697 :size 11597 allpass-composed)
       :where allpass-composed
         (send input :through
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)
           (all-pass :feedback -0.7 :feedforward 0.7)))))

(define-stalin bus (make-bus))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (write-bus bus
                     (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (fx-ctrl input clean wet processor)
  (+ (* clean input)
     (* wet (processor input))))

(<rt-stalin>
  (spawn
    (softsynth))
  (sound
    (out (fx-ctrl (read-bus bus)
                  0.5 0.09
                  reverb))))
```

Signal buses are far from being an "unusual technique", but in text based languages they are in disadvantage compared to graphical music languages such as Max [Puckette, 2002] or Pd. In text based languages it's inconvenient to write to buses, read from buses, and most importantly; it's hard to see the signal flow. However, signal buses (or something which provides similar functionality) are necessary, so it would be nice to have a better way to handle them.

9 Routing Signals with Dynamic Scoping. (Getting rid of manually handling sound buses)

A slightly less verbose way to create, read and write signal buses is to use dynamic scoping to route signals. The bus itself is stored in a coroutine-local variable and created using the ***in*** macro.

Dynamic scoping comes from the fact that ***out*** writes to the bus which was last set up by ***in***. In other words, the scope for the *current bus* (the bus used by ***out***) follows the execution of the program. If ***out*** isn't (somehow) called from ***in***, it will instead write to the bus connected to the soundcard.

For instance, instead of writing:

```
(define-stalin bus (make-bus))

(define-stalin (instr1)
  (sound (write-bus bus 0.5)))

(define-stalin (instr2)
  (sound (write-bus bus -0.5)))

(<rt-stalin>
  (instr1)
  (instr2)
  (sound
    (out (read-bus bus))))
```

we can write:

```
(define-stalin (instr1)
  (sound (out 0.5)))

(define-stalin (instr2)
  (sound (out -0.5)))

(<rt-stalin>
  (sound
    (out (in (instr1)
             (instr2)))))
```

What happened here was that the first time ***in*** was called in the main block, it spawned a new coroutine and created a new bus. The new coroutine then ran immediately, and the first thing it did was to change the *current bus* to the newly created bus. The ***in*** macro also made sure that all ***sound*** blocks called from within the ***in*** macro (i.e. the ones created in *instr1* and *instr2*) is going to run before the main ***sound*** block. (That's how ***sound*** coroutines are stored in a tree)

When transforming the MIDI soft synth to use ***in*** instead of manually handling buses, it will look like this:

[13] as implemented by Bill Schottstaedt in the file "jc-reverb.scm" included with Snd. The *fx-ctrl* function is a copy of the function *fxctrl* implemented in Faust's Freeverb example.

```
;; <The reverb instrument is unchanged>

;; Don't need the bus anymore:
(define-stalin bus (make-bus))

;; softsynth reverted back to the previous version:
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
         (sound (out (* (midi-vol)
                        (oscil :freq (midi->hz (midi-note)))))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

;; A simpler main block:
(<rt-stalin>
  (sound
    (out (fx-ctrl (in (softsynth))
                  0.5 0.09
                 reverb))))
```

10 CPS Sound Generators. (Adding stereo reverb and autopanning)

Using coroutine-local variables was convenient in the previous examples. But what happens if we want to implement autopanning and (a very simple) stereo reverb, as illustrated by the graph below?

```
              +-- reverb -> out ch 0
             /
softsynth--<
             \
              +-- reverb -> out ch 1
```

First, lets try with the tools we have used so far:

```
(define-stalin (stereo-pan input c)
  (let* ((sqrt2/2 (/ (sqrt 2) 2))
         (angle (- pi/4 (* c pi/2)))
         (left  (* sqrt2/2 (+ (cos angle) (sin angle))))
         (right (* sqrt2/2 (- (cos angle) (sin angle)))))
    (out 0 (* input left))
    (out 1 (* input right))))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (sound
          (stereo-pan (* (midi-vol)
                         (oscil :freq (midi->hz (midi-note))))
                      (/ (midi-note) 127.0))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(<rt-stalin>
  (sound
    (in (softsynth)
        (lambda (sound-left sound-right)
          (out 0 (fx-ctrl sound-left  0.5 0.09 reverb))
          (out 1 (fx-ctrl sound-right 0.5 0.09 reverb))))))
```

At first glance, it may look okay. But the reverb will not work properly. The reason is that auto-generated variables used for coroutine-local variables are identified by their position in the source. And since the code for the reverb is written only one place in the source, but used two times from the same coroutine, both channels will use the same coroutine-local variables used by the reverb; a delay, four comb filters and four all-pass filters.

There are a few ways to work around this problem. The quickest work-around is to recode 'reverb' into a macro instead of a function. However, since neither the problem nor any solution to the problem are very obvious, plus that it is slower to use coroutine-local variables than manually allocating them (it requires extra instructions to check whether the data has been allocated[14]), it's tempting not to use coroutine-local variables at all.

Instead we introduce a new concept called CPS Sound Generators, where CPS stands for Continuation Passing Style. [Sussman and Steele, 1975]

10.1 How it works

Working with CPS Sound Generators are similar to Faust's Block Diagrams composition. [Orlarey et al., 2004] A CPS Sound Generator can also be seen as similar to a Block Diagram in Faust, and connecting the CPS Sound Generators is quite similar to Faust's Block Diagram Algebra (BDA).

CPS Sound Generators are CPS functions which are able to connect to other CPS Sound Generators in order to build up a larger function for processing samples. The advantage of building up a program this way is that we know what data is needed before starting to process samples. This means that auto-allocated variables don't have to be stored in coroutines, but can be allocated before running the *sound* block.

For instance, the following code is written in generator-style and plays a 400Hz sine sound to the sound card:

```
(let ((Generator (let ((osc (make-oscillator :freq 400)))
                   (lambda (kont)
                     (kont (oscil osc))))))
  (sound
    (Generator (lambda (sample)
                 (out sample)))))
```

The variable *kont* in the function *Generator* is the continuation, and it is always the last argument in a CPS Sound Generator. A continuation is a function containing the rest of the program. In other words, a continuation function

[14] It is possible to optimize away these checks, but doing so either requires restricting the liberty of the programmer, some kind of JIT-compilation, or doing a whole-program analysis.

will never return. The main reason for programming this way is for generators to easily return more than one sample, i.e have more than one output.[15]

Programming directly this way, as shown above, is not convenient, so in order to make programming simpler, some additional syntax have been added. The two most common operators are Seq and Par, which behave similar to the '**:**' and ',' infix operators in Faust.[16]

Seq creates a new generator by connecting generators in sequence. In case an argument is not a generator, a generator will automatically be created from the argument.

For instance, (Seq (+ 2)) is the same as writing

```
(let ((generator0 (lambda (arg1 kont0)
                    (kont0 (+ 2 arg1)))))
  (lambda (input0 kont1)
    (generator0 input0 kont1)))
```

and (Seq (+ (random 1.0)) (+ 1)) is the same as writing

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input kont2)
    (generator0 input (lambda (result0)
                        (generator1 result0 kont2)))))

;; Evaluating ((Seq (+ 2) (+ 1)) 3 display)
;; will print 6!
```

Par creates a new generator by connecting generators in parallel. Similar to *Seq*, if an argument is not a generator, a generator using the argument will be created automatically.

For instance, (Par (+ (random 1.0)) (+ 1)) is the same as writing:

```
(let ((generator0 (let ((arg0 (random 1.0)))
                    (lambda (arg1 kont0)
                      (kont0 (+ arg0 arg1)))))
      (generator1 (lambda (arg1 kont1)
                    (kont1 (+ 1 arg1)))))
  (lambda (input2 input3 kont1)
    (generator0 input2
      (lambda (result0)
        (generator1 input3
          (lambda (result1)
            (kont1 result0 result1)))))))

;; Evaluating ((Par (+ 2)(+ 1)) 3 4 +) will return 10!
```

(gen-sound :*options* generator) is the same as writing

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0)
           (out 0 result0)))))
```

...when the generator has one output. If the generator has two outputs, it will look like this:

```
(let ((gen generator))
  (sound :options
    (gen (lambda (result0 result1)
           (out 0 result0)
           (out 1 result1)))))
```

...and so forth.

The Snd-RT preprocessor knows if a variable or expression is a CPS Sound Generator by looking at whether the first character is capitol. For instance, (Seq (Process 2)) is equal to (Process 2), while (Seq (process 2)) is equal to (lambda (input kont) (kont (process 2 input))), regardless of how 'Process' and 'process' are defined.

10.2 Handling auto-allocated variables

oscil and the other CLM generators are macros, and the expanded code for *(oscil :freq 440)* looks like this:

```
(oscil_ (autovar (make_oscil_ 440 0.0)) 0 0)
```

Normally, *autovar* variables are translated into coroutine-local variables in a separate step performed after macro expansion. However, when an auto-allocated variable is an argument for a generator, the *autovar* surrounding is removed. And, similar to other arguments which are normal function calls, the initialization code is placed before the generator function. For example, (Seq (oscil :freq 440)) is expanded into:[17]

```
(let ((generator0 (let ((var0 (make_oscil_ 440 0.0)))
                    (lambda (kont)
                      (kont (oscil_ var0 0 0))))))
  (lambda (kont)
    (generator0 kont)))
```

[15] Also note that by inlining functions, the Stalin scheme compiler is able to optimize away the extra syntax necessary for the CPS style.

[16] Several other special operators are available as well, but this paper is too short to document all of them.

[17] Since the Snd-RT preprocessor doesn't know the number of arguments for normal functions such as *oscil_*, this expansion requires the preprocessor to know that this particular *Seq* block has 0 inputs. The preprocessor should usually get this information from the code calling *Seq*, but it can also be defined explicitly, for example like this: (Seq 0 Cut (oscil :freq 440)).

10.3 The Soft Synth using CPS Sound Generators

```
(define-stalin (Reverb)
  (Seq (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (all-pass :feedback -0.7 :feedforward 0.7)
       (Sum (comb :scaler 0.742 :size 9601)
            (comb :scaler 0.733 :size 10007)
            (comb :scaler 0.715 :size 10799)
            (comb :scaler 0.697 :size 11597))
       (delay :size (* .013 (mus-srate)))))

(define-stalin (Stereo-pan c)
  (Split Identity
         (* left)
         (* right)
    :where left  (* sqrt2/2 (+ (cos angle) (sin angle)))
    :where right (* sqrt2/2 (- (cos angle) (sin angle)))
    :where angle (- pi/4 (* c pi/2))
    :where sqrt2/2 (/ (sqrt 2) 2)))

(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (define tone
        (gen-sound
          (Seq (oscil :freq (midi->hz (midi-note)))
               (* (midi-vol))
               (Stereo-pan (/ (midi-note) 127)))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (stop tone))))))

(define-stalin (Fx-ctrl clean wet Fx)
  (Sum (* clean)
       (Seq Fx
            (* wet))))

(<rt-stalin>
  (gen-sound
    (Seq (In (softsynth))
         (Par (Fx-ctrl 0.5 0.09 (Reverb))
              (Fx-ctrl 0.5 0.09 (Reverb))))))
```

11 Adding an ADSR Envelope

And finally, to make the MIDI soft synth sound decent, we need to avoid clicks caused by suddenly starting and stopping sounds. To do this, we use a built-in ADSR envelope generator (entirely written in Scheme) for ramping up and down the volume. Only the function *softsynth* needs to be changed:

```
(define-stalin (softsynth)
  (while #t
    (wait-midi :command note-on
      (gen-sound :while (-> adsr is-running)
        (Seq (Prod (oscil :freq (midi->hz (midi-note)))
                   (midi-vol)
                   (-> adsr next))
             (Stereo-pan (/ (midi-note) 127))))
      (spawn
        (wait-midi :command note-off :note (midi-note)
          (-> adsr stop)))
      :where adsr (make-adsr :a 20:-ms
                             :d 30:-ms
                             :s 0.2
                             :r 70:-ms))))
```

12 Conclusion

This paper has shown a few techniques for doing low-latency sample-by-sample audio programming.

13 Acknowledgments

Thanks to the anonymous reviewers and Anders Vinjar for comments and suggestions.

References

William Clinger and Jonathan Rees. 1991. Revised Report (4) On The Algorithmic Language Scheme.

Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408.

O.-J. Dahl and K. Nygaard. 1966. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678.

Kjetil Matheussen. 2008. Realtime Music Programming Using Snd-RT. In *Proceedings of the International Computer Music Conference.*

Kjetil Matheussen. 2009. Conservative Gargage Collectors for Realtime Audio Processing. In *Proceedings of the International Computer Music Conference*, pages 359–366 (online erratum).

James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(2):61–68.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust, *soft computing.*

Miller Puckette. 2002. Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

W. Schottstaedt. 1994. Machine Tongues XVII: CLM: Music V Meets Common Lisp. *Computer Music Journal*, 18(2):30–37.

Jr. Steele, Guy Lewis and Gerald Jay Sussman. 1978. The Revised Report on SCHEME: A Dialect of LISP. *Technical Report 452, MIT.*

Gerald Jay Sussman and Jr. Steele, Guy Lewis. 1975. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab.*

Ge Wang and Perry Cook. 2003. ChucK: a Concurrent and On-the-fly Audio Programming Language, *Proceedings of the ICMC.*