Simulating a garbage collected heap in Matlab/TrueTime

Technical report

Sven Gestegård Robertz
Department of Computer Science
Lund University

April 2006

Abstract

Languages with automatic memory management have made their entrance on the real-time and control systems scene, and recent development in research on scheduling of garbage collection has produced different approaches to real-time garbage collection. As different systems have different real-time requirements, there is not one approach to garbage collection scheduling that performs best in all cases. Rather, there are trade-offs between predictability and throughput, memory usage and CPU overhead, etc.

To facilitate the study of the interaction between real-time and memory requirements, a memory management simulator has been developed using the Matlab/Simulink-based simulator TrueTime. The simulator allows simulation of a complete computer-controlled physical system, with co-simulation of an entire control system, including the memory system, the real-time kernel and the process under control.

1 Introduction

The memory simulator was written to allow experimentation with GC scheduling and studies of how memory management costs affect both real-time and control performance to be carried out in a controlled environment. Having a model of a garbage collected heap in the Matlab/TrueTime environment allows co-simulation of a continous-time dynamic system (process), a real-time kernel, and a memory system. As both the tasks controlling the process and the garbage collector are executed on the simulated TrueTime kernel, a complete control system can be simulated.

The fundamental approach to the GC simulation described here is to use an abstracted heap representation to record the set of objects (live, dead and free) on the heap, a GC work metric to determine the CPU time required to perform a GC cycle given a certain heap state, and a TrueTime task performing garbage collection. The simulation is not intended for detailed study of the performance of GC algorithms, but the model is generic enough to allow comparisons between how different types of algorithms perform on a certain application.

The GC simulation is based on a generic heap model and a mark-sweep garbage collector. The heap model is driven by the mutator's allocation of objects and pointer assignmnent, and the GC is used to determine the number of live and dead objects (the mark routine) or to reclaim memory (sweep). The GC executes in a separate task. In each invocation of the GC task, the heap state is measured, and the GC work function is evaluated, and when the execution time of the GC task is equal or greater than the total work of that cycle (according to the work function), the cycle finishes. This makes the simulation fairly accurate, as the actions of the mutator affects the CG workload of the current cycle. That also means that the scheduling will affect the amount of floating garbage: With a concurrent GC, if the GC gets much CPU time early in the cycle, and finishes early, less objects will have had time to die.

1.1 TrueTime

The simulator is implemented on top of TrueTime, a Matlab and Simulink-based system for studying embedded control systems by co-simulating the timing properties of a real-time kernel and the continous time dynamics of the process under control [HCÅ02].

TrueTime makes it possible to simulate the temporal behavior of a multi-tasking kernel and to study how the timing affects the performance controller tasks. Tasks are implemented as Matlab m-files, or in C++. Processes under control are modeled as ordinary Simulink blocks. Different scheduling policies may be used, e.g., priority or deadline based scheduling. The execution times of tasks can be constant or time-varying, using some suitable probability distribution. The effects of context switches and interrupts are taken into account, as well as blocking due to task synchronization using events and monitors. TrueTime also includes network blocks to simulate timing effects of different network protocols, which makes it possible to simulate distributed systems.

Figure 1 shows a screenshot of the memory simulator. The physical process and the control computer are simulink blocks, and both the states of the process and different signals in the computer, e.g., the schedule, amount of free memory, GC cycle time and execution time, etc, are available as Simulink signals.

1.2 GC work metrics

The calculation of the GC work required to complete a GC cycle is based on a GC work function, or GC metric [Hen98, Rob06]. That allows simulation of any tracing GC, by simply changing the work function. The work metric function has the signature

```
work = gc_metric(livecount, livebytes, liverefs, deadcount, deadbytes)
```

Equation 1 shows the work function for a basic mark-sweep collector, and Equation 2 describes the cost of a mark-compact algorithm. In both cases, the terms with *livecount* and $liverefs$ represent the cost of marking objects, and the term with *deadbytes* represents object initialisation.

$$W_{MS} = \alpha \cdot livecount + \beta \cdot liverefs + \gamma_c \cdot deadcount + \gamma_s \cdot deadbytes \quad (1)$$

$$W_{MC} = \alpha_c \cdot livecount + \beta \cdot liverefs + \alpha_s \cdot livebytes + \gamma_s \cdot deadbytes; \quad (2)$$

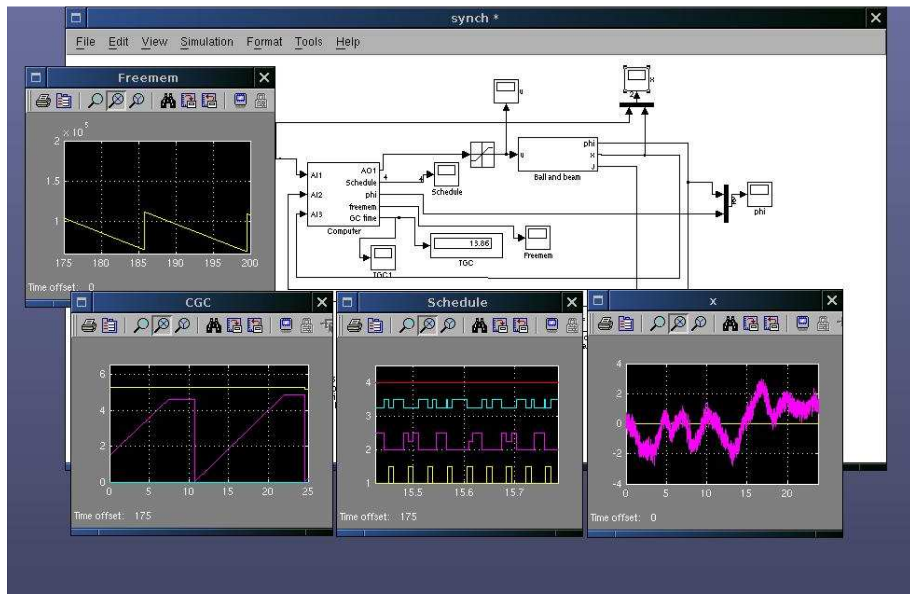

Figure 1: Screenshot of the TrueTime-based memory simulator.

Thus, these metrics allow modelling the effects of object size distribution and pointer density, but not the actual placement of objects on the heap. Factors in the GC work that depend on object placement have to be calculated based on assumptions on e.g. the probability that an object will have to be moved or that free space can be coalesced.

2 Design

2.1 Data structures

The heap is represented by a matrix where each row (or "slot") represents one object. Slots representing allocated objects are on the format
`[size, mark, nptrs, ptr1, ... , ptrn, don't care, ... ]`,
and slots representing free space have the form
`[0, next_free, don't care, ...]`

For the allocated objects,

size is the size in bytes

mark is 1 if marked, 0 otherwise

nptrs is the number of pointers

ptrk is the heap slot pointed to by pointer k

The left over space at the end of the slot, after the last pointer, is undefined.

For the free slots, `next_free` is the index of the next slot in the freelist, or `0` if this is the last. The rest of the vector elements, at the end of the slot, are

undefined. As all matrix rows are of the same length; the "real" size is indicated by the size element. slotlength must be set to accomodate the largest slot.

For example, if an object can contain at most four pointers, the slots would look like in Table 1. A simple object graph is shown in Figure 2, and an example of a corresponding heap is given in Table 2.

size	mark	2	p1	p2	-	-
0	next_free	-	-	-	-	-

Table 1: Layout of the heap representation; the first row shows an allocated object, the second row shows a free slot.

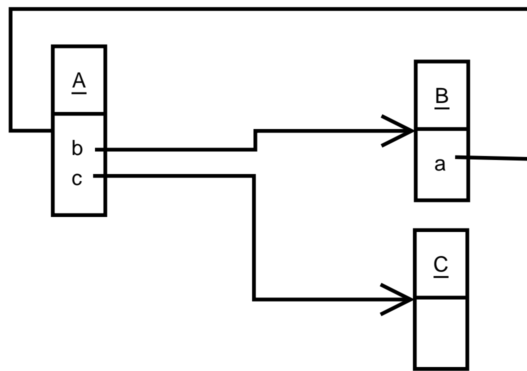

Figure 2: Simple object graph

The two variables `free` and `used` are used to record the actual memory usage, and the locations of free slots are recorded using a free-list, to facilitate allocation.

64	mark	2	4	6	-	-
0	3	-	-	-	-	-
0	5	-	-	-	-	-
32	mark	1	1	-	-	-
0	0	-	-	-	-	-
48	mark	0	-	-	-	-

Table 2: Layout of the heap representation corresponding to Figure 2. Objects are of size 64, 32 and 48 bytes, respectively. The slots 2, 3 and 5 are free, and the `freelist` variable points to the first free slot, 2.

2.2 Operation

The GC simulation is based on the assumption that the amount of GC work can be sufficiently well approximated by a gc metric:

$$C_{\text{GC}} = f(livecount, livebytes, liverefs, deadcount, deadbytes \quad (3)$$

for some function f.

Internally in the simulation, a batch mark-sweep collector is used. The mark phase (which can be run at any time, and any number of times) is used to obtain the heap state.

Given the heap state, any (tracing) GC algorithm can be simulated using its corresponding metric. For instance, if we want to simulate a concurrent mark-copy collector, at each increment we call `gc_mark` to get the current heap state, and insert this into the gc_metric function to obtain the total GC work (in increments) required to complete a GC cycle. When this number is equal to the number of increments performed during the current cycle, the cycle is finished, and the total execution time is the number of increments times the time of a single increment.

When the simulated GC has completed its cycle, GC is performed on the heap representation by calling `gc_sweep`.

Floating garbage occurs in the simulated system for two reasons. First, in the `gc_sweep` routine, a probability (`floatprob`) for that an unmarked object will float for one cycle, can be set. Second, as mentioned, the scheduling of the GC task affects the amount of floating garbage; if the CPU utilization is low and the GC finishes long before its deadline, less objects have time to die and the amount of floating garbage increases.

The current implementation assumes that the simulated GC only makes memory available to the mutator at the end of each cycle, as a moving or copying collector does. This has the drawback that it cannot simulate GCs that continously free memory, e.g. traditional incremental mark-sweep. From a real-time point of view, however, this is not a significant limitation as relying on using memory being freed before the deadline of the GC means borrowing memory from the following GC cycle. While this might help avoid an out-of-memory situation due to too slow GC progress, it means that the real-time performance of the mutator relies on the placement of objects in memory. From a robustness point of view, that is unacceptable. Thus, the limitation does not affect the simulation results, with relation to schedulability.

3 Usage

This section describes how mutator and collector tasks are implemented.

3.1 Mutator operations

The operations available to the mutator is the standard set: object allocation, field access and pointer update. Additionally, the root stack push and pop are required to facilitate stack scanning.

Technically, there are no stacks, per se; all references reside on the heap [1]. From a user perspective, root operations behave almost as expected, with the difference that the object that will become a root must be allocated before it is pushed on the root stack.

```
listhead = allocate(12,3);
push_root(listhead);
```

Field access ($a.b$) is done through the `ptr(a,b)` function. Pointer update ($a.b = c$) is done with `ptr_assign(a,b,c)`. Continuing the example, the object referenced by `listhead` represents a singly linked list with the first element referenced by field 3. The following code allocates an object of size `elem_size` and inserts it at the beginning:

```
tmp = allocate(elem_size,2);
first = ptr(listhead,3);
ptr_assign(tmp,1,first);
ptr_assign(data.listhead,3,tmp);
```

In the above example, the field is accessed using filed numbers. Symbolic names can be introduced by using variables:

```
listhead_thelist = 3;
...
first = ptr(listhead, listhead_thelist);
```

3.2 Collector operations

The collector implementation is based on tracing the heap to determine the heap state, and using the `gc_metric` function to calculate the total amount of GC work required to complete the current cycle. For each invocation of the `gc_task` function, one GC increment is performed, and when `totalwork` increments have been performed, the GC cycle is completed.

The TrueTime task representing the GC, `gc_task` models a concurrent GC. In the following, a time-triggered GC [RH03] is asssumed. I.e., the scheduling is controlled by assigning a period time and a deadline to the GC task. Thus, in a system with scheduling parameters calculated off-line, the parameters are set when the GC task is created. In principle, an adaptive scheduler would do the following:

```
TGC = calculateTGC();
ttSetPeriod(TGC,'gc_task');
ttSetDeadline(GC, 'gc_task');
```

A snippet from the TrueTime `gc_task` illustrates the operation of the simulation. When the GC is running, the following code is executed in each invocation (corresponding to one TrueTime segment)

[1] The reason for this is to avoid having to implement a reference data structure. Instead, references are simply indices in the heap vector.

```
gc_progress = gc_progress + 1;

[livecount, livebytes, liverefs] = gc_mark;
deadcount = alloccount-(livecount+floatingcount);
deadbytes = used-(livebytes+floatingbytes);

totalwork = gc_metric(livecount, livebytes, liverefs, deadcount, deadbytes);
if gc_progress > totalwork
  gc_state = 2; % GC cycle finished
end
exectime = gc_time_inc;
```

The variable `gc_time_inc` holds the time (in CPU seconds) one atomic GC increment takes. Thus, when the GC cycle finishes, the execution time of the GC task is $gc_progress \cdot gc_time_inc$ CPU seconds.

After enough work required to complete the GC cycle has been performed, the actual reclamation of memory is done by calling `gc_sweep()` and `gc_flip`:

```
[deadcount, deadmem] = gc_sweep;
gc_flip;
```

If desired, the flip can be delayed until the following GC cycle is triggered.

4 Conclusion

The aim of this tool to allow co-simulation of both real-time and control performance of an embedded control system with automatic memory management. It is not intended for performing detailed study on the performance of a particular GC algorithm implementation.

The strength of the tool is that the model of the simulated GC algorithm does not require implementation of GC algorithms, but only a GC work function to be identified. That makes it easy to add or change the GC algorithms used for simulation, while providing reasonably realistic simulation. As the GC work function is evaluated at each GC invocation, the actual scheduling affects the amount of GC work, the amount of floating garbage, etc.

The main weakness of the described approach is speed: the overhead of tracing the heap in order to find the heap state is significant, especially if very fine-grained simulation (i.e., very short GC increments) is required. Thus, there is a trade-off between simulation granularity and simulation speed. Also, the simulation does not offer instruction-level correctness, as it is based on approximative models.

A API

This section lists the different functions of the simulator. The format of the listing is as follows, with two or three lines per function.

```
file name
signature, if the function takes parameters or returns a value
description
```

First, the operations used by the mutator is presented, and then the ones used by the collector.

A.1 Mutator API

```
allocate.m
allocate(size, nptrs)
Perform an allocation, returns a pointer to the new object

pop_root.m
Pop a root from the root stack

ptr.m
res = ptr(a, b)
performs field access : return a.b ;

ptr_assign.m
ptr_assign(a, b, c)
performs pointer assignment: a.b = c;

push_root.m
Push a root on the root stack
```

A.2 GC

```
gc_flip.m
Perform a flip (or corresponding). I.e., start a new GC cycle

gc_mark.m
function [livecount, livemem, refcount] = gc_mark()
Perform the mark phase

gc_metric.m
gcwork = gc_metric(livecount, livebytes, liverefs, deadcount, deadbytes)
Compute the amount of GC work (in increments) required for a certain
heap state, for mark-sweep, mark-copy and copying collection, respectively
The result, gcwork, is a vector containing the result of the gc work function
for each of the supported GC algorithms.
```

```
gc_sweep.m
[deadcount, deadmem] = gc_sweep()

memman.m
initialize the heap
```

References

[HCÅ02] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[RH03] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, California, USA, June 2003. ACM Press.

[Rob06] Sven Gestegård Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Department of Computer Science, Lund University, 2006.