Scheduling Garbage Collection in Real-time Systems

Martin Kero
Martin.Kero@ltu.se

Simon Aittamaa
Simon.Aittamaa@ltu.se

Department of Computer Science
Luleå University of Technology
Luleå, Sweden

ABSTRACT

The key to successful deployment of garbage collection in real-time systems is to enable provably safe schedulability tests of the real-time tasks. At the same time one must be able to determine the total heap usage of the system. Schedulability tests typically require a uniformed model of timing assumptions (inter-arrival times, deadlines, etc.). Incorporating the cost of garbage collection in such tests typically requires both artificial timing assumptions of the garbage collector and restricted capabilities of the task scheduler. In this paper, we pursue a different approach. We show how the reactive object model of the programming language Timber enables us to decouple the cost of a concurrently running copying garbage collector from the schedulability of the real-time tasks. I.e., we enable *any* regular schedulability analysis without the need of incorporating the cost of an interfering garbage collector. We present the *garbage collection demand analysis*, which determines if the garbage collector can be feasibly scheduled in the system.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and embedded systems]; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Algorithms, Languages, Reliability, Verification

Keywords

Schedulability analysis, reactive systems

1. INTRODUCTION

As capabilities of embedded devices increase, the possibilities and demands of more complex systems arise. This leads to an ever increasing need of more sophisticated runtime system features. The ability to use shared dynamic data and parallel/concurrent execution of tasks are both examples of features that enables the programmer to more extensively utilize these new capabilities. However, the two features in conjunction makes automatic memory management (garbage collection) a necessity.

Garbage collection has widely been acknowledge for reducing development time as well as enhancing reliability of software systems. In the real-time system context, the work on garbage collection can roughly be divided into two categories. The first category concerns the development of garbage collection algorithms suitable for real-time systems. This line of work was initiated already in the late 70's by Baker [2], and has since then received most of the attention. The second category, to which this paper belongs, concerns schedulability analyses of garbage collected real-time systems. This line of research was initiated by Henriksson [14].

A real-time system is defined in terms of a set of *tasks* for which timing assumptions are uniformly defined (inter-arrival times and patterns, deadlines, etc.). Corresponding schedulability tests crucially depend on the uniformed task model [21]. Adapting such tests to incorporate the cost of garbage collection typically imposes unnecessary restrictions on both task model and scheduler. The main issue is that schedulability analysis requires execution time estimates of the tasks to be independent, whereas such estimate of a garbage collector depends on a combination of memory and timing behaviors of the real-time tasks.

In this paper, we pursue a different approach. We show how an incremental copying garbage collector [18] deployed as a concurrent process in the reactive environment of the programming language Timber [23] enables us to decouple the cost of garbage collection from the feasibility of the real-time tasks. I.e., we enable *any* schedulability analysis for a real-time system, without the need of incorporating the cost of garbage collection into that analysis. Instead, we offer another analysis, decoupled from the regular schedulability analysis. Given a schedulable set of tasks, our analysis provides a sufficient test to determine if the garbage collector can be scheduled in the system. We call this analysis *garbage collection demand analysis*. Our approach relies on the following two key properties:

1. Each atomic increment of the garbage collector can be executed within a small and constant time (including root-set scanning and synchronization operations).

2. In order to preserve schedulability of the task set, the garbage collector runs as the lowest priority process.

CODES+ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

The running time of a concurrent copying garbage collector depends on the amount of live heap memory and the synchronization work due to interrupting mutators. Besides timing assumptions for the task set, our analysis requires upper-bounds on global live heap space and heap allocation of each task as input parameters. Such analyses does not fit within the scope of this paper. However, a forthcoming paper presenting live heap space analysis for real-time systems is in preparation [19].

The general motivation behind this paper is pervaded by the ambition to achieve provable schedulability properties of a real-time garbage collector, as opposed to improve performance. Although improving schedulability is important, it is not within the scope of this paper to embark that field. Instead, our objective is to enable schedulability tests for garbage collection independently of which scheduling policy that is employed for the real-time tasks.

2. TIMBER – REACTIVE OBJECTS

Timber is a, strongly typed, object-oriented programming language for reactive real-time systems. The basis of the language is the *reactive object*. This section is a brief description of the parts of the language that are relevant to the rest of this paper. More in-depth descriptions can be found at the Timber official homepage [23], in the draft language report [5], the formal semantics definition [7], and the descriptions of the reactive object model [25, 24].

The reactive objects in Timber are concurrent. That means, activity in one object can execute concurrently with activity in other objects. Furthermore, objects are the only state carrying data structures (mutable data) and the only way to access the state of an object is through its methods. State integrity is preserved by restricting methods of one particular object to execute under mutual exclusion. In order to preserve aliveness, methods cannot block-and-wait for future events. Events are instead interpreted as method calls and each method will eventually terminate, leaving the object in an idle state. Methods can either be asynchronous (equivalent to events in their own right) or synchronous rendezvous operations.

Timber offers a capability to express timing constraints directly in the model. Asynchronous methods (events) are associated with both a *baseline* (earliest release time) and a *deadline* (latest response time). If not explicitly set, the *baseline* of a method invocation is inherited from the method who called it. If the event is due to an interrupt, the baseline is set to the time-stamp of the interrupt. Deadlines are expressed relative to the current baseline. Similar to baselines, deadlines are inherited if not explicitly set.

Correct timing behavior of a method is that it must start executing and finish within its permissible execution window. Baselines and deadlines can be adjusted by explicitly expressing it in the definition of an asynchronous method (or by the caller).

Aside from the reactive objects, Timber consists of a expression layer based on a *pure* (no side-effects, only immutable data) call-by-value functional language.

3. THE TIMBER RUN-TIME MODEL

In order to meet the semantic specification of the language, a Timber program will depend on the infrastructure provided by the run-time kernel of the language. Conceptually, the kernel has to offer the following services:

- Create an object,
- Send a synchronous message, and
- Send an asynchronous message (possibly delayed).

In reality, creating an object maps to allocation of storage. The only difference between allocating an object and an immutable data structure (struct, tuple, cons cell, etc.) is that in order to enable mutual exclusion between methods of the same object, each object needs a *lock* field. A synchronous rendezvous is simply accomplished by locking the object before running the code of the method, and release the lock afterwards. An asynchronous method call on the other hand requires a new *message* (conceptually an execution thread) to be allocated and enqueued in the proper message queue (timer queue if delayed). Once this message gets to run, it simply makes a regular synchronous call.

The default scheduling mechanism in the kernel is a preemptive priority scheduler based on deadlines. Delayed messages are scheduled based on baselines. The scheduler interacts with three types of data; objects (includes locks), messages (including executable code, possibly parameters, a baseline, and a deadline), and threads (messages extended with an execution context). Objects are the shared resources for which messages acquire and release locks. Asynchronous messages that are latent (either delayed or pending) are, when scheduled to run, promoted to a unique thread. The concurrency of a Timber program is thus accomplished by the scheduler; which, for each independent schedulable message, may allocate a new unique thread of execution.

The interface of the scheduler consists of four entry points.

1. Whenever a new asynchronous message is posted (either internally by another method or externally by an interrupt).
2. When a method calls lock or unlock for an object.
3. When a method terminates.
4. When a timer interrupt occurs.

The scheduler manages three main data structures. A stack of active threads (including, of course, each thread's individual stored execution context), a priority queue of pending messages, and a queue of delayed messages ordered by earliest baseline first.

There are a few very important properties of the kernel that are worth mentioning. All lock operations eventually return, either with the acquired lock or with a deadlock error. The lock of an object may only be owned by one message at a time. The highest priority thread, or the thread holding a lock wanted by the highest priority thread, will always be the one scheduled to run. All messages will run after their baselines since a delayed message will be promoted to the queue of pending messages once its baselines has expired. The aliveness property of a Timber program is crucially dependent upon the fact that a successful lock is always followed by an unlock and that all locks are acquired in a nested manner.

4. THE GC ALGORITHM

We will base our analysis on the incremental copying garbage collector presented in [18]. In this section we give a shortened description of the essential parts of the algorithm. For a complete specification with correctness proofs see the original presentation.

We use Cheney's in-place breadth-first traversal of gray objects [9], and we deploy a read barrier similar to that of Brooks [6]; i.e., reads to old copies in white space are forwarded to their corresponding new copies in the gray/black heap. Furthermore, we use a write barrier in the style of Steele [27], where the tri-color invariant is upheld by reverting black objects to gray upon mutation.

Let x, y, z range over heap addresses, and let n range over integers. Let u, v range over values and be either a heap address or an integer. Let U and V range over sequences of such values.

A heap node can be either a sequence of values (enclosed by angle brackets $\langle V \rangle$) or a single forwarding address (denoted $\bullet x$). A heap H is a finite mapping from addresses to nodes, as captured by the following grammar.

$$\begin{array}{lllll} \text{(heap)} & H & ::= & \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\} \\ \text{(node)} & o & ::= & \langle V \rangle \mid \bullet\, x \\ \text{(value)} & v & ::= & x \mid n \end{array}$$

The domain $dom(H)$ of a heap $H = \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}$ is the set $\{x_1, \ldots, x_n\}$. A heap look-up is defined as $H(x) = o$ if $x \mapsto o \in H$. We write U, V to denote the concatenation of the value sequences U and V. Along the same line, we write H, G for the concatenation of heaps H and G, provided their domains are disjoint.

The heap is described as a triple of subheaps separated by heap borders (|). A heap border has the same meaning as the regular concatenation operator for heaps, but it also provides necessary bookkeeping information. The three subheaps capture the white, gray, and black part of the heap as in the *tricolor abstraction* [10].

The algorithm is based on a labeled transition system (LTS) [22], where garbage collection transitions are so called internal (τ) transitions. Each individual τ transition constitutes an atomic increment by the garbage collector. In Figure 1 and 2, all possible internal transitions are shown. Determinism between different internal transitions are achieved by pattern matching, as each configuration matches only one single clause. The clauses are furthermore divided into two groups, which we call *scan* and *copy* transitions. A garbage collection cycle is a sequence of such transitions beginning with a START transition and ending with a DONE transition.

$$H_0 \xrightarrow{\text{START}} H_1 \longrightarrow \ldots \longrightarrow H_{n-1} \xrightarrow{\text{DONE}} H_n$$

Notice that an active garbage collection cycle is identified by a non-empty white subheap.

In contrast, the external transitions, such as mutations and allocations, are labeled, denoted by $H \xrightarrow{l} H'$. The definition of these transitions are shown in Figure 3. We use a single root pointer r to capture the root-set. Even though a real system most likely will contain more than one root, this can easily be captured in our model by adjusting the content of the node pointed to by r. I.e., the actual root-set is the content of the node labelled r.

At the beginning of a cycle the heap has the form $\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$, and initiating a garbage collection cycle (START) invalidates the whole heap except for the root node. That is, all nodes but r are made white by placing them to the left of the white-gray heap border. The algorithm then proceeds by scanning gray nodes (SCANSTART) and takes proper actions when embedded addresses are encountered. This is accomplished by a *scan pointer* that traverses the gray nodes. The scan pointer is denoted by the symbol $\downarrow$ and has similar function and purpose as the heap borders, i.e. regular concatenation as well as bookkeeping information. When a whole node has been scanned it is promoted from gray to black (SCANDONE). When there are no more gray nodes to scan the garbage collector is finished (DONE).

During scanning of a gray node, encountering an address may result in one of three possible actions. If the address found is not in the white heap, the algorithm just goes on to examine the next gray node field (SCANADDR). If the address is in the white heap, and the corresponding node is a forwarding node, the forwarding address replaces the encountered address (FORWARD). If, on the other hand, the node found is a regular white node, copying is initiated (COPYSTART). This is done by allocating a new empty node in the gray heap and *locking* the scan pointer, which we denote by the alternative concatenation symbol symbol $\uparrow_z$ (where the index is the address of the new empty node). The white node is then copied word by word (COPYWORD) until the whole node has been copied (COPYDONE). At this point, the address of the newly allocated node replaces the old encountered address, and the original white node is converted into a forwarding node.

5. SCHEDULING THE GC

The problem of scheduling garbage collection in real-time systems is twofold. The requirements put on the garbage collector scheduler are:

1. *the garbage collector must not cause any task to miss its deadline, and*
2. *the system must not run out of memory.*

One can easily see that fulfilling one of the requirements may cause a failure to meet the other. E.g., to avoid running out of memory, the garbage collector needs to run, which in turn may cause a task to fail meeting its deadline. This scheduling problem is not easily solved and it becomes even more difficult in our case because we use a copying collector, which, conceptually, has to finish before any garbage memory can be reused.

5.1 Idle time GC

In the general case, finding and scanning the root-set incrementally is a very difficult task. The problem is that the root-set is not constant. Finding all roots requires scanning of, not only static fields and CPU registers, but also the run-time stacks. Since the depth and content of the run-time stacks are not constant, the cost of scanning them is bound to be unpredictable. Scanning them incrementally is also deemed to be notoriously hard due to their volatile nature. In the worst case, the scanning process would need to be restarted at each increment since the content of the stacks may have been altered altogether. The idea of possibly restart the scanning process of an arbitrary sized root-set

START	$\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$	$\longrightarrow$	$G \mid r \mapsto \langle V \rangle \mid \emptyset$	
SCANSTART	$W \mid G, x^\dagger \mapsto \langle V \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V \rangle \mid B$	$W \neq \emptyset$, †may be dirty
SCANINT	$W \mid G, x \mapsto \langle V \downarrow n, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, n \downarrow V' \rangle \mid B$	$W \neq \emptyset$
SCANADDR	$W \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, y \downarrow V' \rangle \mid B$	$W \neq \emptyset$, $y \notin dom(W)$
SCANRESTART	$W \mid G, \dot{x} \mapsto \langle V \downarrow V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V, V' \rangle \mid B$	$W \neq \emptyset$
SCANDONE	$W \mid G, x \mapsto \langle V \downarrow \rangle \mid B$	$\longrightarrow$	$W \mid G \mid x \mapsto \langle V \rangle, B$	$W \neq \emptyset$
DONE	$W \mid \emptyset \mid B$	$\longrightarrow$	$\emptyset \mid B \mid \emptyset$	

Figure 1: Scan transitions.

FORWARD	$W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \bullet z, W' \mid G, x \mapsto \langle V, z \downarrow V' \rangle \mid B$	
COPYSTART	$W, y^\dagger \mapsto \langle U \rangle, W' \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U \rangle, W' \mid z \mapsto \langle\rangle, G, x \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty z is fresh
COPYWORD	$W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^\dagger \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U, u, U' \rangle, W' \mid G, z \mapsto \langle U, u \rangle, G', x^\dagger \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty
COPYRESTART	$W, \dot{y} \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U' \rangle, G', x^\dagger \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle\rangle, G', x^\dagger \mapsto \langle V \uparrow_z y, V' \rangle \mid B$	†may be dirty
COPYDONE	$W, y \mapsto \langle U \rangle, W' \mid G, z \mapsto \langle U \rangle, G', x^\dagger \mapsto \langle V \uparrow_z y, V' \rangle \mid B$ $\longrightarrow$ $W, y \mapsto \bullet z, W' \mid G, z \mapsto \langle U \rangle, G', x^\dagger \mapsto \langle V, z \downarrow V' \rangle \mid B$	†may be dirty

Figure 2: Copy transitions.

at each increment is not very appealing, especially not when it comes to real-time systems.

In order to remedy this problem, we will restrict the garbage collector to only run at idle time. Although this policy has been considered inferior to other approaches (in terms of performance), it comes with a couple of substantial advantages.

5.1.1 Constant time root-set scanning

In a reactive system, such as Timber, the root-set during idle time is very small. The queue of pending messages is empty and no run-time stacks exist. In fact, the root-set consists of only two elements, the interrupt vector and the queue of delayed messages. The size of these data structures are directly connected to the number of tasks defined. Since periodic task releases are achieved by posting a delayed message for the next instance of the task, each periodic task will contribute with one message in the queue of delayed messages. Tasks that are connected to external events (hardware interrupts) has their corresponding messages in the interrupt vector. Thus, if the set of tasks is statically known, the size of the root-set is also statically known.

5.1.2 Schedulability analysis is preserved

Since the garbage collector is the lowest priority process in the system it will never compete for CPU time with the real-time tasks. Nonetheless, if the garbage collector is running (which must be assumed in the worst case) some extra overhead is put on the tasks due to synchronization issues. Furthermore, even though the garbage collector is incremental, the longest atomic increment must be taken into account as an execution time overhead for each task.

In the next section we will look into the details of these overheads and show that the synchronization overhead, is very small (zero for the hard real-time case).

6. GC OVERHEAD

Restricting the garbage collector to only run at idle time reduces the problem of determining the time overhead due to garbage collection significantly. The induced overhead can be divided into two parts, the cost of actually inter-

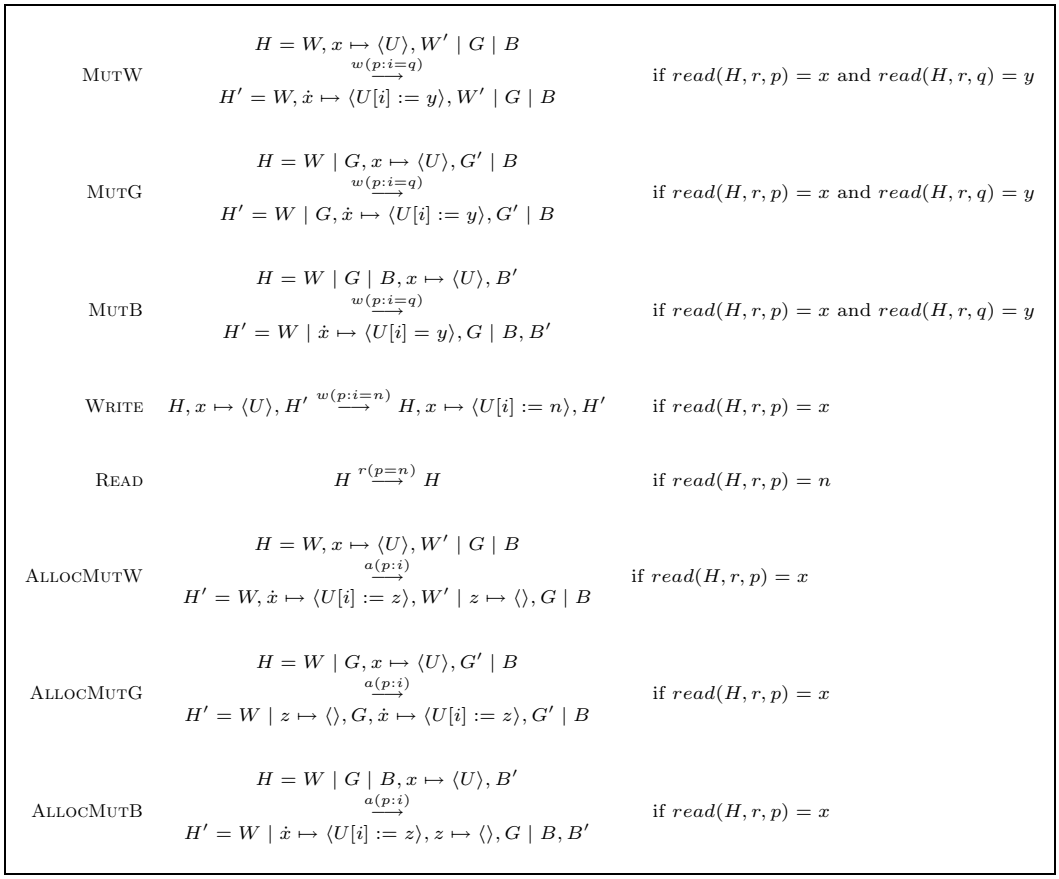

Figure 3: Mutator transitions.

rupting the garbage collector while it is running, and the cost of synchronizing with garbage collector (read and write barriers).

6.1 Longest atomic increment of the GC

The longest atomic increment of the garbage collector is easily determined directly from the algorithm definition (Fig. 1 and 2). Since none of the transitions are dependent on any variable-sized data the cost is constant in terms of number of instructions to execute.

6.2 Synchronization

Generally, the need of synchronization for an incremental garbage collector is to preserve the tri-color invariant [10]. However, for a copying garbage collector the situation is a little bit trickier. Not only is the synchronization mechanism needed to preserve the liveness view of the heap but also for assuring that mutators access the new copies (if they exist) of heap objects. The synchronization between mutators and the garbage collector typically boils down to so called *barrier* methods. One for reading and one for writing heap data.

The reactive objects of Timber enable us to reduce these synchronizations substantially. The reason is that the only mutable data structures are the objects. Immutable data, as pointed out already by Doligez and Leroy [11] as well as Huelsbergen and Larus [15], require no synchronization actions.

Since access to objects must be made under mutual exclusion, a combined read/write barrier can be used that is invoked as part of the lock operation. That is, instead of calling a barrier method for every heap access, we now only need to invoke the barrier when an object is locked. Since most methods only lock one object, the combined read/write barrier is called only once per method invocation.

6.3 Hard real-time systems

Since an object should contain at least one method in order to be meaningful, the concept of dynamic object creation ultimately leads to dynamic creation of tasks. However, generally speaking, in order to determine schedulability, dynamic creation of tasks cannot be allowed. All tasks must be statically known. Thus creating objects dynamically cannot be allowed. These mutable objects can then be allocated static. What is left in the dynamic heap is now only immutable data which do not require any synchronization. Synchronization overhead is hence eliminated altogether. In Fig. 4, an example of how such a system is arranged in memory is shown.

7. SCHEDULABILITY OF THE GC

We have so far shown that the first requirement of garbage collection scheduling is fulfilled by our collector. We have accomplished this by restricting the garbage collector to only

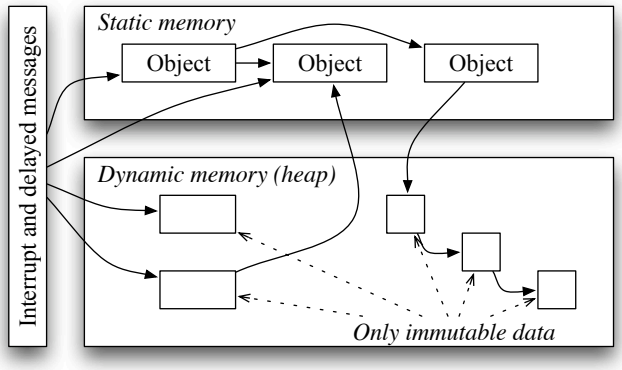

Figure 4: Example of a Timber system with only static objects.

run at idle time. In this section we will look at what is required in order to fulfill the second requirement.

7.1 Garbage collection demand analysis

Baruah et al. presents a feasibility test for real-time tasks called *processor demand analysis* [4, 3]. We present a feasibility test for garbage collection in real-time systems called *garbage collection demand analysis*. In contrast to checking that the processor demand for all possible time windows are less than the size of the window, our analysis determines the required size of the window such that the garbage collection demand is less than the size of the window. If such a window size can be found and the required memory demands of that window size can be met, our garbage collector can feasibly be schedule in the system.

We will only look at the hard real-time case for the analysis, which means no mutable data will be present on the heap and the transitions ScanRestart and CopyRestart will never be taken.

We begin by looking at the execution time of the garbage collector if it is allowed to run without interruptions, and then we look at the cost contributed by tasks interrupting the collector.

The garbage collector transitions are defined in Fig. 1 and 2. The Start and Done transition is only taken once per garbage collection cycle. The number of transitions of each kind can be derived from three basic parameters of the heap to be collected:

M : Amount of reachable memory to copy (in words).
O : Number of reachable nodes on the heap.
R : Number of reachable references on the heap.

We can now formulate the execution time of garbage collection (T_{gc}) in terms of these parameters and execution times of each transition (denoted by $T_{<\text{name of transition}>}$).

$$\begin{aligned} T_{gc} =& M * T_{\text{CopyWord}} + \\ & O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} + T_{\text{CopyStart}} + \\ & T_{\text{CopyDone}}) + R * T_{\text{ScanAddr}} \\ & + (R - O) * T_{\text{Forward}} + T_{\text{Start}} + T_{\text{Done}} \\ =& M * T_{\text{CopyWord}} + \\ & O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} + T_{\text{CopyStart}} + \\ & T_{\text{CopyDone}} - T_{\text{Forward}}) + \\ & R * (T_{\text{ScanAddr}} + T_{\text{Forward}}) \\ & + T_{\text{Start}} + T_{\text{Done}} \end{aligned} \quad (1)$$

Due to the fact that the tasks never can cause anymore copying work for the garbage collector[1] and we only have immutable data on the heap, the only extra garbage collection cost of interruptions is due to new allocations. Similarly to the heap parameters above we have three parameters due to new allocations during garbage collection:

A_i^M : Amount of new memory allocated by task i (in words).
A_i^O : Number of nodes allocated by task i.
A_i^R : Number of references allocated by task i.

We can now formulate the garbage collector execution time due to new allocations made by task i (T_i^A).

$$\begin{aligned} T_i^A =& A_i^O * (T_{\text{ScanStart}} + T_{\text{ScanDone}}) + \\ & A_i^R * T_{\text{ScanAddr}} + \\ & (A_i^R - A_i^O) * T_{\text{Forward}} \\ =& A_i^O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} - T_{\text{Forward}}) + \\ & A_i^R * (T_{\text{ScanAddr}} + T_{\text{Forward}}) \end{aligned} \quad (2)$$

Now we will look at the total garbage collection demand for a time window of size t. For the sake of simplicity we will assume that the heap and allocation parameters are constants. Even though this is not true in reality (reachable memory, allocation rates, etc. tend to vary over time) we can always assume the worst case. We will return to the validity of this assumption at the end of this section.

Let n be the number of tasks, T_i^A be the garbage collection execution time due to allocations made by task i, $\left\lceil \frac{t}{P_i} \right\rceil$ the maximum number of releases of task i in a time window of size t, C_i the execution time of task i, and P_i the period of task i. We can now formulate the total garbage collection demand (C_{gc}) in a time window of size t as follows.

$$C_{gc}(t) = T_{gc} + \sum_{i=1}^{n} \left\lceil \frac{t}{P_i} \right\rceil * (C_i + T_i^A) \quad (3)$$

Informally, C_{gc} consists of two independent parts. One accounting for the execution time required to garbage collect the heap, as if undisturbed. The second part accounts for the extra time induced and consumed by tasks interrupting the garbage collector.

Theorem 7.1. *For any t such that $C_{gc}(t) \leq t$ the garbage collector can be scheduled feasibly (w.r.t. time) with a period of t.*

[1]This is because new data is allocated in tospace and garbage nodes can never become reachable again (see Lemma 5.1 in [18]).

Proof By contradiction. ∎

The memory needs of the system is formulated as a function of the period t of the garbage collector.

$$M_{tot}(t) = 2 * \left(M + \sum_{i=1}^{n} \left\lceil \frac{t}{P_i} \right\rceil * A_i^M \right) \quad (4)$$

In order to complete the proof of correctness of the feasibility test, we need to show that it is sufficient to test feasibility of the worst case of heap and allocation parameters.

LEMMA 7.2. *If the garbage collector can feasibly be scheduled with a period of t for a system with heap parameters M, O, and R, and for all tasks i, allocation parameters A_i^M, A_i^O, and A_i^R. Then, for any $M' \leq M$, $O' \leq O$, $R' \leq R$, and for all tasks i, $A_i^{M'} \leq A_i^M$, $A_i^{O'} \leq A_i^O$, and $A_i^{R'} \leq A_i^R$, the garbage collector can still be feasibly scheduled with a period of t.*

Proof Follows directly from the fact that Equation 1 and 2 are monotonic. ∎

8. EXPERIMENTAL RESULTS

The objective of the experimental study is to confirm that the model conceptually captures the execution time of the garbage collector. In other words, for a particular platform, there shall exist constants (e.g. T_{COPYWORD}) such that we can construct the platform specific model of worst-case garbage collection execution time by simply replacing the constants in the general model by appropriate values. Naturally, experimental results are by no means guaranteed to reveal the worst-case behavior (i.e. the constants associated with the actual worst-case behavior). However, if set up appropriately, it will inevitably expose flaws of the model (non-existence of constants). It should be noted that this experimental study is by no means a performance evaluation of the garbage collector or the way it is scheduled.

In the model, we have five parameters (M, O, R, A_i^O, and A_i^R) that affects the execution time of the garbage collector (T_{gc} and T_i^A). The general approach we pursue is to measure the effect of each parameter in isolation whilst keeping the other parameters constant.

8.1 Hardware platform

The experimental platform we use is the LPC2468 Developer's Kit from Embedded Artists [29], with the standard LPC2468-16 OEM board replaced by an LPC2468-32 OEM board. The LPC2468-32 OEM board contains a standard NXP ARM7TDMI-S LPC2468 microcontroller [30] along with 32 MB of SDRAM from Micron [28]. The reason we choose this platform is simply because it has a sufficient amount of memory (i.e. enables sufficiently wide ranges of input parameter values for the measurements), a JTAG interface, and a fairly simple memory hierarchy. The microcontroller has no cache, but there exists a primitive buffering mechanism within the External Memory Controller (EMC).

8.2 Software platform

In order to make it easy to run the measurements, we use two different code-bases. The first code-base is provided by Embedded Artists [29] and takes care of the initialization of the LPC2468, setting the CPU clock to 57.6 MHz and initializes the External Memory Controller (EMC). When the initialization of the LPC2468 is complete the microcontroller will be stuck in an infinite-loop. Once the infinite-loop is reached we connect to the microcontroller using the JTAG interface and upload the second code-base into the SDRAM. The second code-base consists of the test-code along with the Timber Run-Time System (Timber RTS). The Timber RTS used in the experiments is based on the standard ARM RTS available in the Timber darcs repository [23]. The ARM RTS has then been modified to support logging of several events within the RTS, among other things the GC-time and execution-time of messages. Special care is taken to minimize the effects of the logging. While it is impossible to completely remove the effects of logging the overhead is very small, constant, and predictable. After the test-run is complete we download the log from the internal SRAM of the microcontroller via the serial port.

8.3 Measurements

8.3.1 Parameters affecting T_{gc}

We have three heap parameters affecting the resulting T_{gc}. We setup the test runs in such way that each parameter can be varied in isolation (except when varying number of live nodes). The general configuration is a time-triggered scheduling of the garbage collector. The live data on the heap is a constant structure which is relocated 100 times by the garbage collector for each parameter value. We collect the total running time of each garbage collection cycle. During these measurements, we do not have any tasks running concurrently with garbage collector. We do this procedure for 1000 different values of the parameter in question.

Varying amount of live memory We use one live node, which we vary the size from 1 to 1000 words containing no references. The results of this is shown in Figure 5.

Varying amount of live references We use one live node with a constant size of 1000 words, for which we vary the number of self references from 1 to 1000. The results of this is shown in Figure 6.

Varying number of live nodes We use a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this is shown in Figure 7. Due to the fact that we cannot easily generate equivalent data structures with amount of live memory and number live references constant and vary the number of nodes with sought granularity, we actually vary all three parameters in a controlled way (i.e. 16 words live memory, 1 live reference, and 1 live object per data point).

8.3.2 Parameters affecting T_i^A

We have two heap parameters affecting the resulting T_i^A. We setup the test runs in such way that each parameter can be varied in isolation. The general configuration is, again, a time-triggered scheduling of the garbage collector, but now accompanied with a task interrupting the GC once at every run. The amount of live data on the heap is kept constant over all runs as a constant payload of work to be done. We

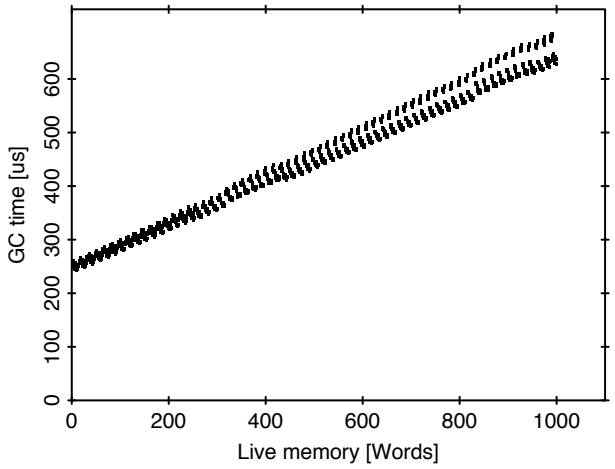

Figure 5: Measurement data of GC time as a function of live memory.

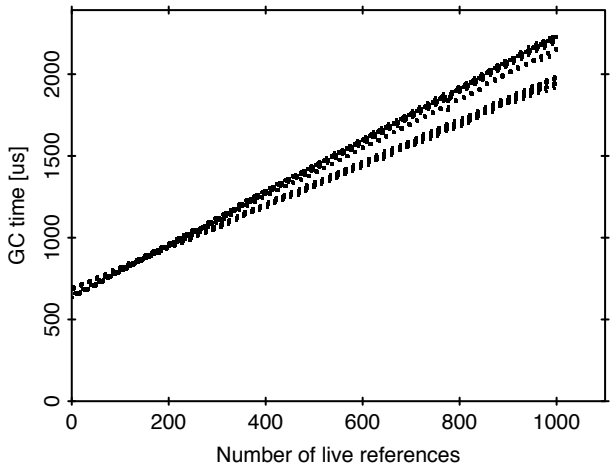

Figure 6: Measurement data of GC time as a function of number of live references.

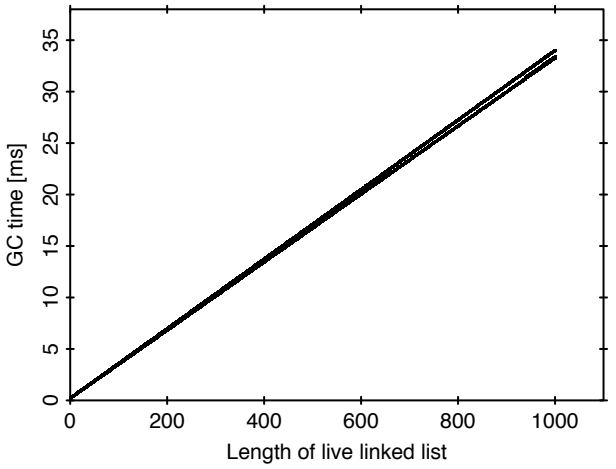

Figure 7: Measurement data of GC time as a function of the length of the live linked list.

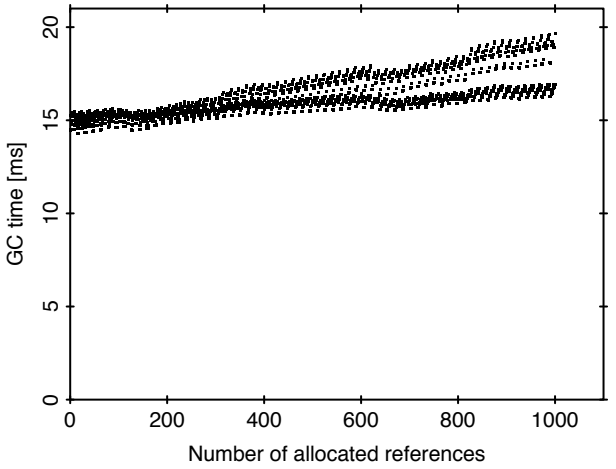

Figure 8: Measurement data of GC time as a function of number of references allocated by an interrupting task.

collect the total running time of each garbage collection cycle. We do this procedure for 1000 different values of the parameter in question.

Varying amount of references allocated We allocate one node with a constant size 1000, for which we vary the number of self references from 1 to 1000. The results of this is shown in Figure 8.

Varying number of nodes allocated We allocate a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this is shown in Figure 9.

From the measurements we can see that the behavior appears to be linear in all parameters. The scattered clustering of data points for some of the measurements can be explained by the, although relatively simple, non-flat behavior of the EMC. The reason why, for some measurements, the clustering forms distinct lines instead of being more evenly distributed in an area is the discrete behavior of the EMC (i.e., either hit or miss in the pre-fetch cache) together with the way the heap parameters were controlled (keeping all but one constant). Nonetheless, the overall tendency is linear which supports the validity of the model.

9. RELATED WORK

The key issue in scheduling concurrent garbage collection in real-time systems is undoubtedly how the garbage collector should be able to compete with the real-time tasks. Typically, this amounts to finding appropriate timing assumptions for the *collector task*. Although the garbage collector does not really have such timing properties, more or less artificial ones are necessary in order to determine schedulability

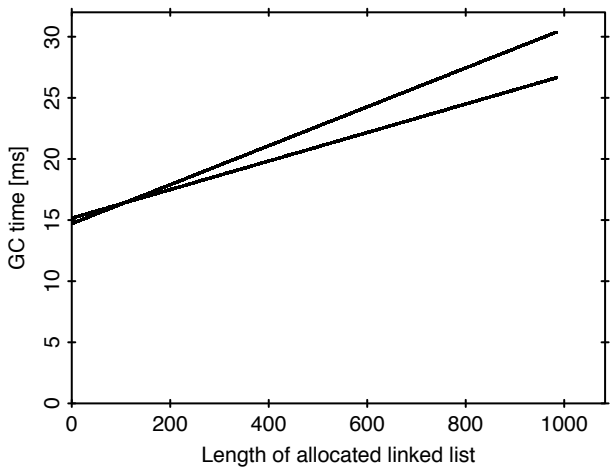

Figure 9: Measurement data of GC time as a function of the length of the linked list allocated by an interrupting task.

of the whole system. Due to the quite complex dependency between properties of the real-time tasks and the required execution time for garbage collection, much of the focus in this field have unfortunately diverged from pure schedulability analysis towards improving measured performance.

In [12], Fu and Hauser presents a framework for describing a broad class of real-time garbage collectors and their corresponding scheduling premises. They also accurately identify the key to enable provable schedulability guarantees of a garbage collected system, to wit comprehensive knowledge about the memory behavior of the real-time tasks as well as the inter-dependencies between them and the garbage collector.

In [13], Henriksson presents a feasibility test for garbage collection based on *response time analysis* [16]. In contrast to our work, he assumes a per task parameter for worst-case garbage collection time required after one invocation (G_i). This parameter slightly corresponds to our execution time parameter due to new allocations (T_i^A). However, he does not present the connection between G_i and the actual garbage collection algorithm used. This work was later extended by Gestegård-Robertz and Henriksson to compute an upper-bound on the cycle time of the garbage collector in order to meet total heap memory limits [26], which corresponds to a rearrangement of our memory needs formula (Equation 4).

In [20], Kim et al. presents upper-bound estimates on the execution time of a garbage collector based on Brook's [6] evacuation strategy. They present a schedulability test for the whole system based on a worst-case response time analysis of a sporadic server. In addition, they also present a live memory analysis to determine the worst-case local live heap memory of each task. In contrast to our work, they do not present a detailed connection between the parameters used in the execution time estimate and the actual garbage collection algorithm used. They are also limited to use rate monotonic priorities to enable the sporadic server schedulability test.

In [8], Chang presents a hybrid approach based on a lazy freeing reference counting collector and a backup mark-sweep collector. External fragmentation is avoided by using a fixed block size. He also presents a schedulability test based on a dual-priority scheduling scheme including the worst-case cost for both the reference counting collector and the mark-sweep collector. This is achieved by integrating the two garbage collectors into the real-time scheduling framework as tasks (i.e., derive appropriate timing assumptions for them). The main difference between his approach and ours is that he integrates the cost of garbage collection in the regular schedulability analysis. This makes it more difficult to extend the regular schedulability test with more features (e.g., shared resources) without affecting the schedulability test of the garbage collector.

Recently, Kalibera et al. developed schedulability tests for both time-based and slack-based scheduling of time-triggered garbage collection [17]. They show that none of them are superior to the other (in terms of schedulability) and they draw the conclusion that the choice of scheduling policy is a key part of designing garbage collected real-time systems.

10. CONCLUSION

We have shown how the reactive object model of Timber enables us to decouple the schedulability analysis of the real-time tasks from the cost of garbage collection. We have, based on the incremental copying garbage collector presented in [18] and the real-time programming language Timber [23], developed a feasibility test for the collector. We call this test *garbage collection demand analysis*, which contains only clearly identified parameters of the real-time system and their corresponding effect on garbage collection time. Apart from formal verification of correctness, we have confirmed the validity of the model through an experimental study run on the LPC2468 Developer's Kit from Embedded Artists.

The key motivation behind our approach to schedule garbage collection is to enable *any* scheduling policy (with corresponding feasibility test) for the real-time tasks. This is achieved by identifying and exploiting key properties of the run-time behavior of real-time tasks defined in Timber, which allow us to decouple the cost of garbage collection from the processor demand of the real-time tasks.

11. FURTHER WORK

Apart from timing assumptions and properties required by regular schedulability analyses (such as inter-arrival times, deadlines, worst-case execution times, etc.) our analysis also requires global live heap space bounds and heap allocation bounds for each task. Analyzing heap allocation properties for each task corresponds quite well to execution time analysis (with a slightly different cost model). Both are monotonically increasing accumulative properties. The results of such program analyses are typically expressed as functions of the programs input data [1]. In a real-time system, where tasks maintain an ongoing interaction with the environment, input data are typically tightly coupled with the state of the system. As a fortunate coincidence, such state-dependent properties corresponds very well to the behavior of global live heap space. Kero et al. has a forthcoming paper in preparation, presenting live heap space analysis for real-time systems [19].

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments.

12. REFERENCES

[1] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *9th International Symposium on Memory management*, 2010.

[2] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990.

[4] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, November 1990.

[5] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science and Engineering, Oregon Health and Science University, April 2002.

[6] R. 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. ACM Press, August 1984.

[7] M. Carlsson, J. Nordlander, and D. Kieburtz. The Semantic Layers of Timber. In *The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. C Springer-Verlag.*, 2003.

[8] Y. Chang. *Garbage Collection for Flexible Hard Real-Time Systems*. PhD thesis, University of York, July 2007.

[9] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[10] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[11] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, 1993.

[12] W. Fu and C. Hauser. A real-time garbage collection framework for embedded systems. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 20–26, New York, NY, USA, 2005. ACM.

[13] R. Henriksson. Predictable Automatic Memory Management for Embedded Systems. In *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, 1997.

[14] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, 1998.

[15] L. Huelsbergen and J. R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *Principles Practice of Parallel Programming*, pages 73–82, 1993.

[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[17] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling hard real-time garbage collection. *Real-Time Systems Symposium, IEEE International*, pages 81–92, 2009.

[18] M. Kero, J. Nordlander, and P. Lindgren. A Correct and Useful Incremental Copying Garbage Collector. In *Proceedings of the 2007 international symposium on Memory Management (ISMM'07)*, Montréal, Québec, Canada, October 2007.

[19] M. Kero, P. Pietrzak, and J. Nordlander. Live heap space bounds for real-time systems. In preparation. `http://staff.www.ltu.se/~keero/LiveHeap.pdf`, 2010.

[20] T. Kim, N. Chang, and H. Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software (JSS)*, 58(3):245–258, September 2001.

[21] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[22] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[23] J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber homepage. http://timber-lang.org, 2008.

[24] J. Nordlander, M. Carlsson, M. Jones, and J. Jonsson. Programming with Time-Constrained Reactions, 2005.

[25] J. Nordlander, M. P. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive Objects. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Arlington, VA, 2002.

[26] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM.

[27] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[28] Webpage. http://download.micron.com/pdf/datasheets/dram/mobile/256MbSDRAMx32_low_power.pdf, April 2010.

[29] Webpage. http://embeddedartists.com/, April 2010.

[30] Webpage. http://www.standardics.nxp.com/products/lpc2000/lpc24xx/, April 2010.