Scheduling of Hard Real-Time Garbage Collection

Martin Schoeberl

Abstract Automatic memory management or garbage collection greatly simplifies development of large systems. However, garbage collection is usually not used in real-time systems due to the unpredictable temporal behavior of current implementations of a garbage collector. In this paper we propose a real-time garbage collector that can be scheduled like a normal real-time thread with a deadline monotonic assigned priority. We provide an upper bound for the collector period so that the application threads will never run out of memory. Furthermore, we show that the restricted execution model of the Safety Critical Java standard simplifies root scanning and reduces copying of static data. Our proposal has been implemented and evaluated in the context of the Java processor JOP.

Keywords Real-Time Systems · Garbage Collection · Real-Time Java

1 Introduction

The Java programming language is widely used for general purpose programming. Java has a number of safety features (with respect to programming errors) which make it an appealing candidate for real-time systems. One key feature that makes Java a safe language is automatic memory management based on a garbage collector (GC). Memory management is a cross-cutting issue and hard to get right when done by hand, especially in large software systems. Manual memory management errors can lead to dangling references which are hard to find and can occur at any point during the execution of a program. Garbage collection relieves programmers from having to worry about this class of errors.

In order to make Java suitable for hard real-time systems the Real-Time Specification for Java (RTSJ) [10] chooses to avoid GC by introducing the concept of immortal

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
E-mail: mschoebe@mail.tuwien.ac.at

and scoped memory areas and no-heap real-time threads. A scoped memory area is a region which supports linear time allocation of objects and bulk deallocation. The RTSJ mandates write barriers to prevent dangling pointers. Any reference assignment must ensure that the referred object has a lifetime at least as long as that of the target of the assignment. Scoped memories are thus safer than explicit memory management, but still hard to use correctly [20, 23]. An alternative that is an active area of research is garbage collection algorithms with real-time guarantees.

We believe that for the acceptance of Java for real-time systems, the restrictions imposed by the RTSJ are too strong. To simplify creation of possible large real-time applications, most of the code should be able to use the GC managed heap. For a collector to be used in real-time systems two points are essential:

- The GC has to keep up with the garbage generated by the application threads to avoid out-of-memory stalls
- The GC has to be incremental with a short maximum blocking time that has to be known
- All GC operations, such as object allocation, write barriers, root scanning, tracing, and object copying need to be time predictable.

The first issue that has to be considered is scheduling the GC so that the GC collects enough garbage. The memory demands (static and dynamic) by the application threads have to be analyzed. These requirements, together with the properties of the GC, result in scheduling parameters for the GC thread. In this paper we present a solution to calculate the maximum period of the GC thread that will collect enough memory in each collector cycle so we will never run out of memory. The collector cycle depends on the heap size and the allocation rate of the application threads.

The second point is necessary to limit interference between the GC thread and high-priority threads. It is also essential to minimize the overhead introduced by read- and write-barriers, which are necessary for synchronization between the GC thread and the application threads. The design of a GC within these constraints is described in [36, 33] and summarized in Section 4.

Time predictability of individual GC operations depends in the implementation of the GC. Although the main focus of this paper is on the derivation of the maximum GC period, the evaluation section provides some details of a time-predictable GC implementation.

To distinguish between other garbage collectors and a collector for (hard) real-time systems we define a real-time collector as follows:

A real-time garbage collector provides time predictable automatic memory management for tasks with a bounded memory allocation rate with minimal temporal interference to tasks that use only static memory.

To avoid heap fragmentation (external or internal) a real-time GC has to compact the heap. A compacted heap results in a time-predictable object allocation. Compaction can be either performed on the full heap with a *mark-compact* collector or with a *copying* collector. In Section 2, the maximum period for both variants is analyzed. As the maximum period is similar for both algorithms and a copying collector

moves less objects we have chosen to implement a copying collector for the evaluation. The collector presented in this paper is based on the work by Steele [38], Dijkstra [14] and Baker [8]. However, the copying collector is changed to perform the copy of an object concurrently by the collector and not as part of the mutator work. Therefore we name it *concurrent-copy* collector.

We will use the terms first introduced by Dijkstra with his *On-the-Fly* concurrent collector [14]. The application is called the *mutator* to reinforce that the application changes (mutates) the object graph while the GC does the collection work. The GC process is simply called *collector*. In the following discussion we will use the color scheme of white, gray, and black objects:

Black indicates that the object and all immediate descendants have been visited by the collector.

Grey objects have been visited, but the descendants may not have been visited by the collector, or the mutator has changed the object.

White objects are unvisited. At the beginning of a GC cycle all objects are white. At the end of the tracing, all white objects are garbage.

At the end of a collection cycle all black objects are live (or floating garbage) and all white objects are garbage.

1.1 Incremental Collection

An incremental collector can be realized in two ways: either by doing part of the work on each allocation of a new object or by running the collector as an independent process. For a single-threaded application, the first method is simpler as less synchronization between the application and the collector is necessary. For a multi-threaded environment there is no advantage by interleaving collector work with object allocation. In this case we need synchronization between the collector work done by one thread and the manipulation of the object graph performed by the other mutator thread. Therefore we will consider a concurrent solution where the collector runs in its own thread or processor. It is even possible to realize the collector as dedicated hardware [15].

1.2 Conservatism

Incremental collector algorithms are conservative, meaning that objects becoming unreachable during collection are not collected by the collector — they are floating garbage. Many approaches exist to reduce this conservatism in the general case. However, algorithms that completely avoid floating garbage are impractical. For different conservative collectors the worst-case bounds are all the same (i.e., all objects that become unreachable during collection remain floating garbage). Therefore the level of conservatism is not an issue for real-time collectors.

1.3 Safety Critical Java

In [35] a profile for safety-critical Java (SCJ) is defined. SCJ is also an evolving standard under the Java Community Process (JSR 302) [16] for future safety-critical systems. SCJ has two interesting properties that may simplify the implementation of a real-time collector: firstly, the split between initialization and mission phase, and secondly the simplified threading model (which also mandates that self-blocking operations are illegal in the mission phase). During initialization of the application a SCJ virtual machine does not have to meet any real-time constraints (other than possibly a worst case bound on the entire initialization phase). It is perfectly acceptable to use a non-real-time GC implementation during this phase – even a stop-the-world GC. As the change from initialization to mission phase is explicit, it is clear when the virtual machine must initiate real-time collection and which code runs during the mission phase.

Simplifying the threading model has the following advantage, if the collector thread runs at a lower priority than all other threads in the system, it is the case that when it runs *all* other threads have returned from their calls to run(). This is trivially true due to the priority preemptive scheduling discipline.[1] Any thread that has not returned from its run() method will preempt the GC until it returns. This has the side effect that the GC will never see a root in the call stack of another thread. Therefore, the usually atomic operation of scanning call stacks can be omitted in the mission phase. We will elaborate on this property in Section 3.

This paper is an extended version of [29] and the remainder of this paper is structured as follows: Section 2 is the main section of the paper and provides the bound of the maximum collector period for mark-compact and concurrent-copy collectors. In Section 3, we describe possible simplifications of the GC algorithm when the application is structured according to the SCJ specification. We describe an implementation of a concurrent-copy GC on the Java processor JOP and evaluate our design in Section 4. It has to be noted that the analysis of the maximum GC period is independent of the concrete implementation. The findings are discussed in Section 5 and compared to related work in Section 6. The paper is concluded in Section 7.

2 Scheduling of the Collector Thread

In the following section, we provide a bound on the maximum collector period so that no mutator thread runs out of memory. To provide this bound, the size of the heap, the maximum allocation rate of the mutator threads, and the maximum lifetime of objects needs to be known. The heap size is known statically. The maximum allocation rate can be analyzed similar to the worst-case execution time (WCET). To derive the maximum allocation rate, a WCET tool, such as [32], can be adapted by setting the execution cost for an allocation instruction to the size of the object and to zero for other instructions. Therefore, the mutator threads need to be statically analyzable, e.g., all loops and the recursion depths needs to be bounded. This is a common restric-

---

[1] If we would allow blocking in the application threads, we would also need to block the GC thread.

tion for hard real-time systems. For the maximum lifetime of objects an inter-thread analysis has to be performed, which is an interesting research challenge.

The collector work can be scheduled either *work* based or *time* based. On a work based scheduling, as performed in [37], an incremental part of the collector work is performed at object allocation. This approach sounds quite natural as threads that allocate more objects have to pay for the collector work. Furthermore, no additional collector thread is necessary. The main issue with this approach is to determine how much work has to be done on each allocation – a non trivial question as collection work consists of different phases. A more subtle question is: Why should a high frequency (and high priority) thread increase its WCET by performing collector work that does not have to be done at that period? Leaving the collector work to a thread with a longer period allows higher utilization of the system.

On a time based scheduling of the collector work, the collector runs in its own thread. Scheduling this thread as a *normal* real-time thread is quite natural for a hard real-time system. The question is: which priority to assign to the collector thread? The Metronome collector [7] uses the highest priority for the collector. Robertz and Henriksson [25] and Schoeberl [29] argue for the lowest priority. When building hard real-time systems the answer must take scheduling theory into consideration: the priority is assigned according to the period, either rate monotonic [19] or more general deadline monotonic [3]. Assuming that the period of the collector is the longest in the system and the deadline equals the period the collector gets the lowest priority.

In this section, we provide an upper bound for the collector period so that the application threads will never run out of memory. The collector period, besides the WCET of the collector, is the single parameter of the collector that can be incorporated in standard schedulability analysis.

The following symbols are used in this section: heap size for a mark-compact collector (H_{MC}) and for a concurrent-copying collector (H_{CC}) containing both semi-spaces, period of the GC thread (T_{GC}), period of a single mutator thread (T_M), period of mutator thread i (T_i) from a set of threads, and memory amount allocated by a single mutator (a) or by mutator i (a_i) from a set of threads.

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst-case,[2] but very common as we can see in the following code fragment.

```
for (;;) {
    Node n = new Node();
    work(n);
    waitForNextPeriod();
}
```

The object Node is allocated at the start of the period and n will reference it until the next period when a new Node is created and assigned to n. In this example we assume that no reference to Node is stored (inside work) to an object with a longer lifetime.

---

[2] See Section 2.3.6 for an example where the worst-case lifetime is two periods.
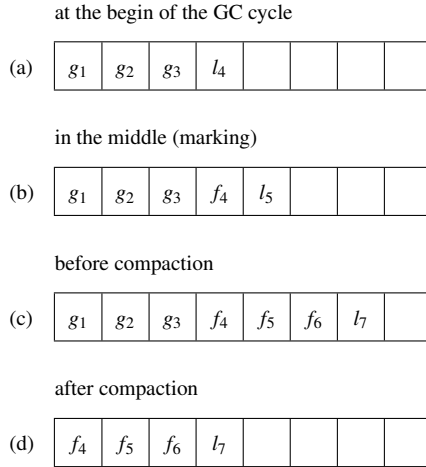
at the begin of the GC cycle

(a) | g_1 | g_2 | g_3 | l_4 | | | | |

in the middle (marking)

(b) | g_1 | g_2 | g_3 | f_4 | l_5 | | | |

before compaction

(c) | g_1 | g_2 | g_3 | f_4 | f_5 | f_6 | l_7 | |

after compaction

(d) | f_4 | f_5 | f_6 | l_7 | | | | |

Fig. 1 Heap usage during a mark-compact collection cycle

at the begin of the GC cycle

(a) | l_4 | g_3 | g_2 | g_1 | | | | | |

from-space   *to-space*

in the middle of the cycle

(b) | f_4 | g_3 | g_2 | g_1 | | f_4 | | | l_5 |

at the end (before flip)

(c) | f_4 | g_3 | g_2 | g_1 | | f_4 | l_7 | f_6 | f_5 |

after the flip

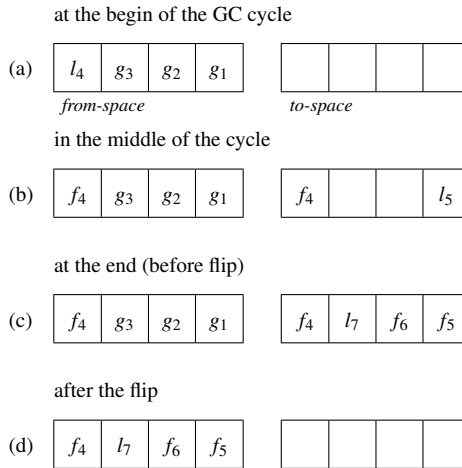(d) | f_4 | l_7 | f_6 | f_5 | | | | | |

Fig. 2 Heap usage during a concurrent-copy collection cycle

2.1 An Example

We start our discussion with a simple example[3] where the collector period is 3 times the mutator period ($T_{GC} = 3T_M$) and a heap size of 8 objects ($8a$). We show the heap during one GC cycle for a mark-compact and a concurrent-copy collector. The following letters are used to show the status of a memory cell (that contains one object from the mutator in this example) in the heap: g_i is garbage from mutator cycle i, l

---

[3] The relation between the heap size and the mutator/collector proportion is an arbitrary value in this example. We will provide the exact values in the next sections.

is the live memory, and f is floating garbage. We assume that all objects that become unreachable during the collection remain floating garbage.

Figure 1 shows the changes in the heap during one collection cycle. At the start (sub-figure (a)) there are three objects (g_1, g_2, and g_3) left over from the last cycle (floating garbage) which are collected by the current cycle and one live object l_4. During the collection the live objects become unreachable and are now floating garbage (e.g. f_4 in sub-figure (b)). At the end of the cycle, just before compacting, we have three garbage cells (g_1-g_3), three floating garbage cells (f_4-f_6) and one live cell l_7 (sub-figure (c)). Compaction moves the floating garbage and the live cell to the start of the heap and we end up with four free cells (sub-figure (d)). The floating garbage will become garbage in the next collection cycle and we start over with the first sub-figure with three garbage cells and one live cell.

Figure 2 shows one collection cycle of the concurrent-copy collector. We have two memory spaces: the *from-space* and the *to-space*. Again we start (sub-figure (a)) the collection cycle with one live cell and three garbage cells left over from the last cycle. Note that the order of the cells is different from the previous example. New cells are allocated in the to-space from the top of the heap, whereas moved cells are allocated from the bottom of the heap. Sub-figure (b) shows a snapshot of the heap during the collection: formerly live object l_4 is already floating garbage f_4 and copied into to-space. A new cell l_5 is allocated in the to-space. Before the flip of the two semi-spaces (sub-figure (c)) the from-space contains the three garbage cells (g_1-g_3) and the to-space the three floating garbage cells (f_4-f_6) and one live cell l_7. Sub-figure (d) shows the heap after the flip: The from-space contains the three floating cells which will be garbage cells in the next cycle and the one live cell. The to-space is now empty.

From this example we see that the necessary heap size for a mark-compact collector is similar to the heap size for a copying collector. We also see that the compacting collector has to move more cells (all floating garbage cells and the live cell) than the copying collector (just the one cell that is live at the beginning of the collection).

2.2 Minimum Heap Size

In this section, we show the memory bounds for a mark-compact collector with a single heap memory and a concurrent-copying collector with the two spaces *from-space* and *to-space*.

2.2.1 Mark-Compact

For the mark-compact collector, the heap H_{MC} can be divided into allocated memory M and free memory F

$$H_{MC} = M + F = G + \overline{G} + L + F \tag{1}$$

where G is garbage at the start of the collector cycle that will be reclaimed by the collector. Objects that become unreachable during the collection cycle and will not

be reclaimed are floating garbage $\overline{G}$. These objects will be detected in the next collection cycle. We assume the worst case that all objects that die during the collection cycle will not be detected and therefore are floating garbage. L denotes all live, i.e., reachable, objects. F is the remaining free space.

We have to show that we will never run out of memory during a collection cycle ($F \geq 0$). The amount of allocated memory M has to be less than or equal to the heap size H_{MC}

$$H_{MC} \geq M = G + \overline{G} + L \tag{2}$$

In the following proof the superscript n denotes the collection cycle. The subscript letters S and E denote the value at the start and the end of the cycle, respectively.

Lemma 1 *For a collection cycle the amount of allocated memory M is bounded by the maximum live data L_{max} at the start of the collection cycle and two times A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + 2A_{max} \tag{3}$$

Proof During a collection cycle G remains constant. All live data that becomes unreachable will be floating garbage. Floating garbage $\overline{G}_E$ at the end of cycle n will be detected (as garbage G) in cycle $n+1$.

$$G^{n+1} = \overline{G}_E^n \tag{4}$$

The mutator allocates A memory and transforms part of this memory and part of the live data at the start L_S to floating garbage $\overline{G}_E$ at the end of the cycle. L_E is the data that is still reachable at the end of the cycle.

$$L_S + A = L_E + \overline{G}_E \tag{5}$$

with $A \leq A_{max}$ and $L_S \leq L_{max}$. A new collection-cycle start immediately follows the end of the former cycle. Therefore the live data remains unchanged.

$$L_S^{n+1} = L_E^n \tag{6}$$

We will show that (3) is true for cycle 1. At the start of the first cycle we have no garbage ($G = 0$) and no live data ($L_S = 0$). The heap contains only free memory.

$$M_S^1 = 0 \tag{7}$$

During the collection cycle the mutator allocates A^1 memory. Part of this memory will be live at the end and the remaining will be floating garbage.

$$A^1 = L_E^1 + \overline{G}_E^1 \tag{8}$$

Therefore at the end of the first cycle

$$M_E^1 = L_E^1 + \overline{G}_E^1$$
$$M^1 = A^1 \tag{9}$$

As $A^1 \leq A_{max}$ (3) is fulfilled for cycle 1.

Under the assumption that (3) is true for cycle n, we have to show that (3) holds for cycle $n+1$.

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{10}$$

$$M^n = G^n + \overline{G}_E^n + L_E^n \tag{11}$$

$$
\begin{aligned}
M^{n+1} &= G^{n+1} + \overline{G}_E^{n+1} + L_E^{n+1} & &\tag{12}\\
&= \overline{G}_E^n + L_S^{n+1} + A^{n+1} & &\text{apply (4) and (5)}\\
&= \overline{G}_E^n + L_E^n + A^{n+1} & &\text{apply (6)}\\
&= L_S^n + A^n + A^{n+1} & &\text{apply (5)} \tag{13}
\end{aligned}
$$

As $L_S \leq L_{max}$, $A^n \leq A_{max}$ and $A^{n+1} \leq A_{max}$

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{14}$$

$\square$

2.2.2 Concurrent-Copy

In the following section, we will show the memory bounds for a concurrent-copying collector with the two spaces *from-space* and *to-space*. We will use the same symbols as in Section 2.2.1 and denote the maximum allocated memory in the from-space as M_{From} and the maximum allocated memory in the to-space as M_{To}.

For a copying-collector the heap H_{CC} is divided in two equal sized spaces H_{From} and H_{To}. The amount of allocated memory M in each semi-space has to be less than or equal to $\frac{H_{CC}}{2}$

$$H_{CC} = H_{From} + H_{To} \geq 2M \tag{15}$$

Lemma 2 *For a collection cycle, the amount of allocated memory M in each semi-space is bounded by the maximum live data L_{max} at the start of the collection cycle and A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + A_{max} \tag{16}$$

Proof Floating garbage at the end of cycle n will be detectable garbage in cycle $n+1$

$$G^{n+1} = \overline{G}_E^n \tag{17}$$

Live data at the end of cycle n will be the live data at the start of cycle $n+1$

$$L_S^{n+1} = L_E^n \tag{18}$$

The allocated memory M_{From} in the from-space contains garbage G and the live data at the start L_s.

$$M_{From} = G + L_S \tag{19}$$

All new objects are allocated in the to-space. Therefore the memory requirement for the from-space does not change during the collection cycle. All garbage G remains in the from-space and the to-space only contains floating garbage and live data.

$$M_{To} = \overline{G} + L \tag{20}$$

At the start of the collection cycle, the to-space is completely empty.

$$M_{To\_S} = 0 \tag{21}$$

During the collection cycle all live data is copied into the to-space and new objects are allocated in the to-space.

$$M_{To\_E} = L_S + A \tag{22}$$

At the end of the collector cycle, the live data from the start L_S and new allocated data A stays either live at the end L_E or becomes floating garbage $\overline{G}_E$.

$$L_S + A = L_E + \overline{G}_E \tag{23}$$

For the first collection cycle there is no garbage ($G = 0$) and no live data at the start ($L_S = 0$), i.e. the from-space is empty ($M_{From}^1 = 0$). The to-space will only contain all allocated data A^1, with $A^1 \leq A_{max}$, and therefore (16) is true for cycle 1.

Under the assumption that (16) is true for cycle n, we have to show that (16) holds for cycle $n+1$.

$$M_{From}^{n+1} \leq L_{max} + A_{max}$$
$$M_{To}^{n+1} \leq L_{max} + A_{max} \tag{24}$$

At the start of a collection cycle, the spaces are flipped and the new to-space is cleared.

$$H_{From}^{n+1} \Leftarrow H_{To}^n$$
$$H_{To}^{n+1} \Leftarrow \emptyset \tag{25}$$

The from-space:

$$M_{From}^n = G^n + L_S^n \tag{26}$$
$$M_{From}^{n+1} = G^{n+1} + L_S^{n+1} \tag{27}$$
$$= \overline{G}_E^n + L_E^n$$
$$= L_S^n + A^n \tag{28}$$

As $L_S \leq L_{max}$ and $A^n \leq A_{max}$

$$M_{From}^{n+1} \leq L_{max} + A_{max} \tag{29}$$

The to-space:

$$M_{To}^n = \overline{G}_E^n + L_E^n \tag{30}$$

$$M_{To}^{n+1} = \overline{G}_E^{n+1} + L_E^{n+1} \tag{31}$$

$$= L_S^{n+1} + A^{n+1}$$

$$= L_E^n + A^{n+1} \tag{32}$$

As $L_E \leq L_{max}$ and $A^{n+1} \leq A_{max}$

$$M_{To}^{n+1} \leq L_{max} + A_{max} \tag{33}$$

□

From this result we can see that the dynamic memory consumption for a mark-compact collector is similar to a concurrent-copy collector. This is contrary to the common belief that a copy collector needs the double amount of memory.

We have seen that the double-memory argument against a copying collector does not hold for an incremental real-time collector. We need double the memory of the allocated data during a collection cycle in either case. The advantage of the copying collector over a compacting one is that newly allocated data are placed in the to-space and does not need to be copied. The compacting collector moves all newly created data (that is mostly floating garbage) at the compaction phase.

2.3 Garbage Collection Period

GC work is inherently periodic. After finishing one round of collection the GC starts over. The important question is: what is the *maximum* period the GC can be run so that the application will never run out of memory. Scheduling the GC at a shorter period does not hurt but decreases utilization.

In the following, we derive the maximum collector period that guarantees that we will not run out of memory. The maximum period T_{GC} of the collector depends on L_{max} and A_{max} for which safe estimates are needed.

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst case, but very common.

2.3.1 Single Mutator Thread

First we give an upper bound for the collector cycle time for a single mutator thread.

Lemma 3 *For a single mutator thread with period T_M that allocates memory "a" each period, the maximum collector period T_{GC} that guarantees that we will not run out of memory is*

$$T_{GC} \leq T_M \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{34}$$

$$T_{GC} \leq T_M \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{35}$$

Proof The maximum live data referenced by a single mutator is the maximum data allocated by the mutator in a single cycle.

$$L_{max} = a \tag{36}$$

A single mutator allocates a memory during the period T_M. Therefore the maximum allocation during the collector period T_{GC} is

$$A_{max} = a \left\lceil \frac{T_{GC}}{T_M} \right\rceil \tag{37}$$

Using equations (2) and (3) we get the minimum heap size H_{MC} for a mark-compact collector

$$H_{MC} \geq L_{max} + 2A_{max}$$
$$H_{MC} \geq a \left(1 + 2 \left\lceil \frac{T_{GC}}{T_M} \right\rceil \right) \tag{38}$$

Equations (15) and (16) result in the minimum heap size H_{CC}, containing both semi-spaces, for the concurrent-copy collector

$$H_{CC} \geq 2(L_{max} + A_{max})$$
$$H_{CC} \geq 2a \left(1 + \left\lceil \frac{T_{GC}}{T_M} \right\rceil \right) \tag{39}$$

The ceiling function covers the worst-case schedule between the collector thread and the mutator thread. We are interested in the maximum collector period T_{GC} with a given heap size H_{MC} or H_{CC}

$$\left\lceil \frac{T_{GC}}{T_M} \right\rceil \leq \frac{H_{MC} - a}{2a} \tag{40}$$

$$\left\lceil \frac{T_{GC}}{T_M} \right\rceil \leq \frac{H_{CC} - 2a}{2a} \tag{41}$$

The maximum quotient ($\frac{T_{GC}}{T_M}$) that fulfills (40) or (41) is an integer n. n is the largest integer that is less than or equal the right side of (40) or (41). Therefore we get for the mark-compact collector

$$\frac{T_{GC}}{T_M} \leq \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{42}$$

$$\Rightarrow T_{GC} \leq T_M \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{43}$$

and for the concurrent-copy collector

$$\frac{T_{GC}}{T_M} \leq \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{44}$$

$$\Rightarrow T_{GC} \leq T_M \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{45}$$

$\square$

2.3.2 Several Mutator Threads

In this section, the upper bound of the period for the collector thread is given for n independent mutator threads.

Theorem 1 *For "n" mutator threads with period T_i where each thread allocates a_i memory each period, the maximum collector period T_{GC} that guarantees that we will not run out of memory is*

$$T_{GC} \leq \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{46}$$

$$T_{GC} \leq \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{47}$$

Proof For n mutator threads with periods T_i and allocations a_i during each period the values for L_{max} and A_{max} are

$$L_{max} = \sum_{i=1}^{n} a_i \tag{48}$$

$$A_{max} = \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{49}$$

The ceiling function for A_{max} covers the individual worst cases for the thread schedule and cannot be solved analytically. Therefore we use a conservative estimation A'_{max} instead of A_{max}.

$$A'_{max} = \sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1 \right) a_i \geq \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{50}$$

From (2) and (3) we get the minimum heap size for a mark-compact collector

$$H_{MC} \geq L_{max} + 2A_{max}$$

$$\geq \sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{51}$$

For a given heap size H_{MC} we get the conservative upper bound of the maximum collector period T_{GC} [4]

$$2A'_{max} \leq H_{MC} - L_{max}$$

$$2\sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1 \right) a_i \leq H_{MC} - L_{max} \tag{52}$$

$$T_{GC} \leq \frac{H_{MC} - L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{53}$$

---

[4] It has to be noted that this is a conservative value for the maximum collector period T_{GC}. The maximum value $T_{GC_{max}}$ that fulfills (51) is in the interval

$$\left(\frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}}, \frac{H_{MC} - \sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \right)$$

and can be found by an iterative search.

$$\Rightarrow T_{GC} \le \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{54}$$

Equations (15) and (16) result in the minimum heap size H_{CC}, containing both semi-spaces, for the concurrent-copy collector

$$H_{CC} \ge 2L_{max} + 2A_{max}$$

$$\ge 2\sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{55}$$

For a given heap size H_{CC} we get the conservative upper bound of the maximum collector period T_{GC}

$$2A'_{max} \le H_{CC} - 2L_{max}$$

$$2\sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1 \right) a_i \le H_{CC} - 2L_{max} \tag{56}$$

$$T_{GC} \le \frac{H_{CC} - 2L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{57}$$

$$\Rightarrow T_{GC} \le \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{58}$$

$\square$

2.3.3 Producer/Consumer Threads

So far we have only considered threads that do not share objects for communication. This execution model is even more restrictive than the RTSJ scoped memories that can be shared between threads. In this section, we discuss how our GC scheduling can be extended to account for threads that share objects.

Object sharing is usually done by a producer and a consumer thread. I.e., one thread allocates the objects and stores references to those objects in a way that they can be accessed by the other thread. This other thread, the consumer, is in charge to *free* those objects after use.

An example of this sharing is a device driver thread that periodically collects data and puts them into a list for further processing. The consumer thread, with a longer period, takes all available data from the list at the start of the period, processes the data, and removes them from the list. During the data processing, new data can be added by the producer. Note that in this case the list will probably never be completely empty. This typical case cannot be implemented by an RTSJ shared scoped memory. There would be no point in the execution where the shared memory will be empty and can get recycled.

The question now is how much data will be alive in the worst case. We denote T_p as the period of the producer thread τ_p and T_c as the period of the consumer thread

τ_c. τ_p allocates a_p memory each period. During one period of the consumer τ_c the producer τ_p allocates

$$\left\lceil \frac{T_c}{T_p} \right\rceil a_p$$

memory. The worst case is that τ_c takes over all objects at the start of the period and frees them at the end. Therefore the maximum amount of live data for this producer/-consumer combination is

$$\left\lceil \frac{2T_c}{T_p} \right\rceil a_p$$

To incorporate this extended lifetime of objects we introduce a lifetime factor l_i which is

$$l_i = \begin{cases} 1 & : \text{ for normal threads} \\ \left\lceil \frac{2T_c}{T_i} \right\rceil & : \text{ for producer } \tau_i \text{ and associated consumer } \tau_c \end{cases} \tag{59}$$

and extend L_{max} from (48) to

$$L_{max} = \sum_{i=1}^{n} a_i l_i \tag{60}$$

The maximum amount of memory A_{max} that is allocated during one collection cycle is not changed due to the *freeing* in a different thread and therefore remains unchanged.

The resulting equations for the maximum collector period are

$$T_{GC} \leq \frac{H_{MC} - \sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{61}$$

and

$$T_{GC} \leq \frac{H_{CC} - 2\sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \tag{62}$$

If the consumer thread does not consume all elements at one iteration, the lifetime of objects is further extended. The of lifetime factor has to be increased accordingly.

2.3.4 Long Periods

Threads with a longer period/deadline than the collector will receive a lower priority than the collector thread. Due to the ceiling functions the equations for the maximum collector period are still valid: Such a thread contributes with his allocation a_l once to L_{max} and once to A_{max}. As a consequence the collector will run at a higher priority than the mutator with the longer period.

As the collector runs at a higher priority than the thread with the long period, we cannot apply the SCJ optimization for this thread. The collector has to scan the stack of that thread atomically. However, this atomic section needs only be atomic with respect to the thread scanned [24]. The collector can still be preempted by higher priority threads during the stack scan.

2.3.5 Static Objects

The discussion about the collector cycle time assumes that all live data is produced by the periodic application threads and the maximum lifetime is one period. However, in the general case we have also live data that is allocated in the initialization phase of the real-time application and stays alive until the application ends. We incorporate this value by including this static live memory L_s in L_{max}

$$L_{max} = L_s + \sum_{i=1}^{n} a_i l_i \qquad (63)$$

A mark-compact collector moves all static data to the bottom of the heap in the first and second[5] collection cycle after the allocation. It does not have to compact these data during the following collection cycles in the mission phase. The concurrent-copy collector moves these static data in each collection cycle. Furthermore, the memory demand for the concurrent copy is increased by the double amount of the static data (compared to the single amount in the mark-compact collector)[6].

The SCJ application model with an initialization and a mission phase can reduce the amount of live data that needs to be copied (see Section 3).

2.3.6 Object Lifetime

Listing 1 shows an example of a periodic thread that allocates an object in the main loop and the resulting bytecodes. The method waitForNextPeriod() (or wFNP() for short) blocks the periodic thread till its next release.

There is a time between allocation of Node and the assignment to n where a reference to the former Node (from the former cycle) and the new Node (on the operand stack) is live. To handle this issue we can either change the values of L_{max} and A_{max} to accommodate this additional object or change the top-level code of the periodic work to explicitly assign a null-pointer to the local variable n as it can be seen in Listing 4 from the evaluation section. Programming against the SCJ profile avoids this issues (see Section 3).

However, this null pointer assignment is only necessary at the top-level method that invokes waitForNextPeriod and is therefore not as complex as explicit freeing of objects. Objects that are created inside work in our example do not need to be *freed* in this way as the reference to the object gets *lost* on return from the method.

2.3.7 Allocation Cost

A compacting GC introduces some overhead due to the movement of live data. However, this additional cost pays off for real-time systems by two important properties: (1) fragmentation (external or internal) is avoided, and (2) allocation is time-predictable. After compaction the free part of the heap is contiguous. Allocation is

---

[5] A second cycle is necessary as this static data can get intermixed by floating garbage from the first collector cycle.

[6] Or the collector period gets shortened.

Listing 1 Example periodic thread and the corresponding Java bytecodes

```java
public void run() {

    for (;;) {
        Node n = new Node();
        work(n);
        waitForNextPeriod();
    }
}
```

```
public void run();
  Code:
   0:   new #20; //class Node
   3:   dup
   4:   invokespecial #22; //"<init>":()V
   7:   astore_1
   8:   aload_1
   9:   invokestatic  #26; //work:(Node)V
  12:   aload_0
  13:   invokevirtual #30; //wFNP:()Z
  16:   pop
  17:   goto   0
```

a cheap and constant time increment of a pointer by the size of the allocated object. A new allocated object in Java has to be initialized with zero. This initialization can be done at allocation time, resulting in a cost linear to the size of the object. Or the free heap can be zeroed out by the GC at the end of the GC cycle. The second solution shifts cost from the mutator threads to the GC thread. When mutator periods are shorter than the GC period, this cost shift results in a better schedulable system.

3 Safety-Critical Java Simplifications

The restrictions of the computational model for safety critical Java allow for optimizations of the GC. We can avoid atomic stack scanning for roots and do not have to deal with exact pointer finding. Static objects, which would belong into immortal memory in the RTSJ, can be detected by a special GC cycle at transition to the mission phase. We can treat those objects specially and do not need to collect them during the mission phase. This static memory area is automatically sized.

It has to be noted that our proposal is extending JSR 302. Clearly, adding RTGC to SCJ reduces the importance of scopes and would likely relegate them to the small subset of applications where fast deallocation is crucial. Discussing the interaction between scoped memory and RTGC is beyond the scope of this paper.

3.1 Simple Root Scanning

Thread stack scanning is usually performed atomically. Scanning of the thread stacks with a snapshot-at-beginning write barrier [40] allows optimization of the write barri-

ers to consider only field access (putfield and putstatic) and array access. Reference manipulation in locals and on the operand stack can be ignored for a write barrier. However, this optimization comes at the cost of a possible large blocking time due to the atomicity of stack scanning.

A subtle difference between the RTSJ and the SCJ definition is the possibility to use local variables within run(). Although handy for the programmer to preserve state information in locals,[7] GC implementation can greatly benefit from *not* having reference values on the thread stack when the thread suspenses execution.

If the GC thread has the lowest priority and there is no blocking library function that can suspend a real-time thread, then the GC thread will only run when all real-time threads are waiting for their next period – and this waiting is performed after the return from the run() method. In that case the other thread stacks are completely *empty*. We do not need to scan them for roots as the only roots are the references in static (class) variables.

For a real-time GC root scanning has to be exact. With conservative stack scanning, where a primitive value is treated as a pointer, possible large data structures can be kept alive artificially. To implement exact stack scanning we need the information of the stack layout for each possible GC preemption point. For a high-priority GC this point can be at each bytecode (or at each machine instruction for compiling Java). The auxiliary data structure to capture the stack layout (and information which machine register will hold a reference for compiled Java) can get quite large [22] or require additional effort to compute.

With a low-priority GC and the RTSJ model of periodic thread coding with wFNP() the number of GC preemption points is decreased dramatically. When the GC runs all threads will be in wFNP(). Only the stack information for those places in the code have to be available. It is also assumed that wFNP() is not invoked very deep in the call hierarchy. Therefore, the stack hight will be low and the resulting blocking time short.

As mentioned before, the SCJ style periodic thread model results in an empty stack at GC runtime. As a consequence we do not have to deal with exact stack scanning and need no additional information about the stack layout.

3.2 Static Memory

A SCJ copying collector will perform best when all live data is produced by periodic threads and the maximum lifetime of a newly allocated object is one period. However, some data structures allocated in the initialization phase stay alive for the whole application lifetime. In an RTSJ application this data would be allocated in immortal memory. With a real-time GC there is no notion of immortal memory; instead we will use the term *static* memory. Without special treatment, a copying collector will move this data at each GC cycle. Furthermore, the memory demand for the collector increases by the amount of the static data.

---

[7] Using multiple wFNP() invocations for local mode changes can also come handy. The author has used this fact heavily in the implementation of a modem/PPP protocol stack.

As those static objects (mostly) live *forever*, we propose a solution similar to the immortal memory of the RTSJ. All data allocated during the initialization phase (where no application threads are scheduled) is considered potentially static. As part of the transition to the mission phase a *special* collection cycle in a stop-the-world fashion is performed. Objects that are still alive after this cycle are assumed to live forever and make up the *static* memory area. The remaining memory is used for the garbage collected heap.

This static memory will still be scanned by the collector to find references into the heap but it is not collected. The main differences between our static memory and the immortal memory of the RTSJ are:

- The choice of allocation context is implicit. There is no need to specify where an object must be allocated. We do not have to state explicitly which data belongs to the application life-time data. This information is implicitly gathered by the start-mission transition.
- References from the static memory to the garbage collected heap are allowed contrary to the fact in the RTSJ that references to scoped memories, that have to be used for dynamic memory management without a GC, are not allowed from immortal memory.

The second fact greatly simplifies communication between threads. For a typical producer/consumer configuration the container for the shared data is allocated in immortal memory and the actual data in the garbage collected heap. With this *immortal* memory solution the actual L_{max} only contains allocated memory from the periodic threads.

4 Evaluation

To evaluate the proposed real-time GC scheduling a concurrent copy collector is implemented in the context of the Java processor JOP [31]. JOP is a Java processor especially designed for embedded real-time systems. The architecture is optimized for worst-case execution time (WCET) instead of the usual optimization for average case execution time. Execution time of bytecodes is known cycle accurate.

First the internals of GC implementation are described, followed by evaluation experiments on heap usage and the release time jitter of high priority threads. The test setup consists of JOP implemented in an Altera Cyclone FPGA clocked at 100 MHz. The main memory is a 1 MB SRAM with an access time of two clock cycles. JOP is configured with a 4 KB instruction cache and a 1 KB stack cache. No additional data cache is used.

4.1 Implementation

The implemented collector on JOP is an incremental collector [29, 36] based on the copy collector by Cheney [13] and the incremental version by Baker [8]. To avoid

the expensive read barrier in Baker's collector all object copies are performed concurrently by the collector. The collector is concurrent and resembles the collectors presented by Steele [38] and Dijkstra et al. [14]. Therefore we call it the *concurrent-copy* collector.

The collector and the mutator are synchronized by two barriers. A Brooks-style [12] forwarding directs the access to the object either into tospace or fromspace. The forwarding pointer is kept in a separate handle area as proposed in [21]. The separate handle area reduces the space overheads as only one pointer is needed for both object copies. Furthermore, the indirection pointer does not need to be copied. The handle also contains other object related data, such as type information, and the mark list. The objects in the heap only contain the fields and no object header.

The second synchronization barrier is a *snapshot-at-beginning* write-barrier [40]. A snapshot-at-beginning write-barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array.

The whole collector, the new operation, and the write barriers are implemented in Java (with the help of native functions for direct memory access). The object copy operation is implemented in hardware and can be interrupted by mutator threads after each word copied [33]. The copy unit redirects the access to the object under copy, depending on the accessed field, either to the original or the new version of the object.

Although we show the implementation on a Java processor, the GC is not JOP specific and can also be implemented on a conventional processor.

4.1.1 Heap Layout

Figure 3 shows a symbolic representation of the heap layout with the handle area and two semi-spaces, *fromspace* and *tospace*. Not shown in this figure is the memory region for runtime constants, such as class information or string constants. This memory region, although logically part of the heap, is neither scanned, nor copied by the GC. This constant area contains its own handles and all references into this area are ignored by the GC.

To simplify object move by the collector, all objects are accessed with one indirection, called the handle. The handle also contains auxiliary object data structures, such as a pointer to the method table or the array length. Instead of Baker's read barrier we have an additional mark stack which is a threaded list within the handle structure. An additional field (as shown in Figure 3) in the handle structure is used for a free list and a use list of handles.

The indirection through a handle, although a very light-weight read barrier, is usually still considered as a high overhead. Metronome [7] uses a forwarding pointer as part of the object and performs forwarding *eagerly*. Once the pointer is forwarded, subsequent uses of the reference can be performed on the direct pointer until a GC preemption point. This optimization is performed by the compiler.

JOP uses a hardware based optimization for this indirection [30]. The indirection is unconditionally performed in the memory access unit. Furthermore, null pointer checks and array bounds checks are done in parallel to this indirection.

There are two additional benefits from an explicit handle area instead of a forwarding pointer: (a) access to the method table or array size needs no indirection,
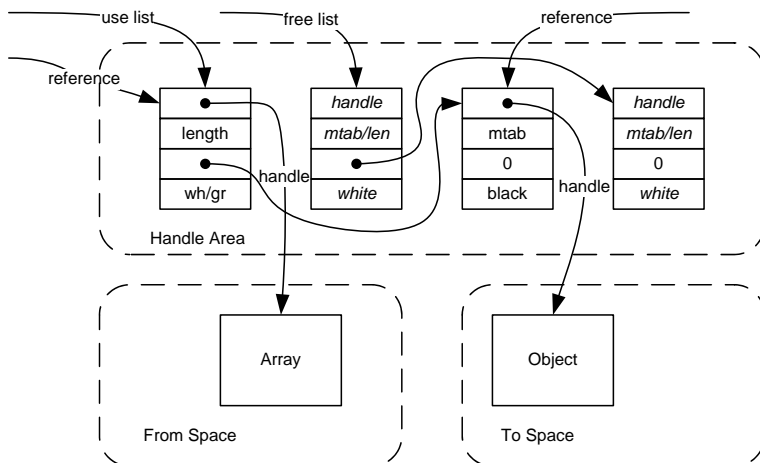
Fig. 3 Heap layout with the handle area

and (b) the forwarding pointer and the auxiliary data structures do not need to be copied by the GC.

The fixed handle area is not subject to fragmentation as all handles have the same size and are recycled at a sweep phase with a simple free list. However, the reserved space has to be sized (or the GC period adapted) for the maximum number of objects that are live or are floating garbage.

4.1.2 The Collector

The collector is scheduled periodically with a priority according to its deadline, which is equal the period. The collector performs following steps within each period it :

Flip   An atomic flip exchanges the roles of tospace and fromspace.

Mark roots   Atomic scan of each thread stack. If the SCJ profile is used this step can be omitted for all higher priority threads, as the thread stacks are empty when the GC executes.

Mark and copy   An object is popped from the mark stack, all referenced objects, which are still white, are pushed on the mark stack, the object is copied to tospace and the handle pointer is updated.

Sweep handles   All handles in the use list are checked if they still point into tospace (black objects) or can be added to the handle free list.

Clear fromspace   At the end of the collector work the fromspace that contains only white objects is initialized with zero. Objects allocated in that space (after the next flip) are already initialized and allocation can be performed in constant time.

Object copy, when performed atomically, can introduce considerable blocking time. This is especially an issue for large arrays. In the implementation on JOP the

blocking time is reduced by a hardware unit the performs the object copy [33]. This unit can be preempted after each copied word. The unit is notified on the interrupt and records the copy position. On an object or array access the hardware knows whether the access should go to the already copied part in the tospace or in the not yet copied part in the fromspace. For a GC on a standard hardware splitting larger arrays into smaller chunks, as done in Metronome [7] and in the GC for the JamaicaVM [37], is a software option to reduce the blocking time.

The collector has two modes of operation: one for the initialization phase and one for the mission phase. At the initialization phase it operates in a stop-the-world fashion and gets invoked when a memory request cannot be satisfied. In this mode the collector scans the stack of the single thread.

As part of the mission start one stop-the-world cycle is performed to clean up the heap from garbage generated at initialization. From that point on the GC runs in concurrent mode in its own thread and can omit scanning of the thread stacks.

4.1.3 The Mutator

The coordination between the mutator and the collector is performed within the new and newarray bytecodes and within write barriers for JVM bytecodes putfield and putstatic for reference fields, and bytecode aastore. The field access bytecodes are substituted at application link time (run of JOPizer). Only write accesses to reference fields are substituted by special versions of the bytecodes (putfield_ref and putstatic_ref). Therefore, the write barrier code is only executed on reference write access.

Allocation Objects are allocated black (in tospace). In non real-time collectors it is more common to allocate objects white. It is argued [14] that objects die young and the chances are high that the GC never needs to touch them. However, in the worst case no object that is created and becomes garbage during the GC cycle can be reclaimed. Those floating garbage will be reclaimed in the next GC cycle. Therefore, we do not benefit from the white allocation optimization in a real-time GC. Allocating a new object black has the benefit that those objects do not need to be copied. The same argument applies to the chosen write barrier. The code in Listing 2 shows the simple implementation of bytecode new.

As the old fromspace is cleared by the GC, the new object is already initialized and new executes in constant time. The methods Native.rdMem() and Native.wrMem() provide direct access to the main memory. Only those two native methods are necessary for an implementation of a GC in pure Java.

Atomic operations, as shown for the implementation of bytecode new, of the GC are protected simply by turning the timer interrupt off.[8] Those atomic sections lead to release jitter of the real-time tasks and shall be minimized. It has to be noted that the GC protection with interrupt disabling is not an option for multiprocessor systems.

---

[8] If interrupt handlers are allowed to change the object graph those interrupts also need to be disabled.

Listing 2 Implementation of bytecode new in JOPs JVM (simplified)

```
private static int new(int cons) {

    int size = ... // read size from the class info

    synchronized (GC.mutex) {

        // we allocate from the upper part
        allocPtr -= size;
        int ref = getHandle(size);
        // mark as object
        Native.wrMem(IS_OBJ, ref+OFF_TYPE);
        // pointer to method table in the handle
        Native.wrMem(cons+CLASS_HEADR, ref+OFF_MTAB_ALEN);
    }

    return ref;
}
```

Listing 3 Snapshot-at-beginning write-barrier in JOPs JVM

```
private static void putfield_ref(int ref, int value, int index) {

    synchronized (GC.mutex) {

        // snapshot-at-beginning barrier
        int oldVal = Native.getField(ref, index);
        // Is it white?
        if (oldVal != 0
            && Native.rdMem(oldVal+GC.OFF_SPACE) != GC.toSpace) {
            // Mark grey
            GC.push(oldVal)
        }
        Native.putField(ref, index, value);
    }
}
```

Write Barriers A snapshot-at-beginning write-barrier [40] synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array. The *to be overwritten* field is shaded gray as shown in Listing 3. An object is shaded gray by pushing the reference of the object onto the mark stack.[9] Further scanning and copying into tospace – coloring it black – is left to the GC thread. One field in the handle area is used to implement the mark stack as a simple linked list. Listing 3 shows the implementation of putfield for reference fields.

Note that field and array access is implemented in hardware on JOP. Only write accesses to reference fields need to be protected by the write-barrier, which is im-

---

[9] Although the GC is a copying collector a mark stack is needed to perform the object copy in the GC thread and not by the mutator.

plemented in software. During class linking all write operations to reference fields (putfield and putstatic when accessing reference fields) are replaced by a JVM internal bytecodes (e.g., putfield_ref) to execute the write-barrier code as shown before. The shown code is part of a special class (com.jopdesign.sys.JVM) where Java bytecodes that are not directly implemented by JOP can be implemented in Java [28].

The methods of class Native are JVM internal methods needed to implement part of the JVM in Java. The methods are replaced by regular or JVM internal bytecodes during class linking. Methods getField(ref, index) and putField(ref, value, index) map to the JVM bytecodes getfield and putfield. The method rdMem() is an example of an internal JVM bytecode and performs a memory read. The null pointer check for putfield_ref is implicitly performed by the hardware implementation of getfield that is executed by Native.getField(). The hardware implementation of getfield triggers an exception interrupt when the reference is null. The implementation of the write-barrier shows how a bytecode is substituted by a special version (pufield_ref), but uses in the software implementation the hardware implementation of that bytecode (Naitve.putfield()).

4.1.4 Time-predictability of GC Operations

The execution time of GC related operations shall be time-predictable and constant at best. Searching for a large enough memory block on allocation is at best hard to analyze. The implemented GC compacts the heap and clears the fromspace at the end of the GC cycle. Therefore, a new operation, as shown in Listing 2, is a constant time operation. The method getHandle() picks the first element from the list of free handles.

Another GC related operation is the write barrier on a reference write (putfield). This operation, as shown in Listing 3, is also a constant time operation. Marking an object grey is performed by pushing it on the mark stack, which itself is also a constant time operation.

4.2 Scheduling Experiments

In this section, we test an implementation of the concurrent-copy garbage collector on JOP. The tests are intended to get some confidence that the formulas for the collector periods are correct. Furthermore, we visualize the actual heap usage of a running system.

The examples are synthetic benchmarks that emulate worst-case execution time (WCET) by executing a busy loop after allocation of the data. The WCET of the collector was measured to be 10.4 ms when executing it with scheduling disabled during one collection cycle for example 1 and 11.2 ms for example 2. We use 11 ms and 12 ms respectively as the WCET of the collector for the following examples[10].

Listing 4 shows our worker thread with the busy loop. The data is allocated at the start of the period and freed after the simulated execution. waitForNextPeriod blocks until the next release time for the periodic thread.

---

[10] It has to be noted that measuring execution time is not a safe method to estimate WCET values.

Listing 4 Example periodic thread with a busy loop

```
public void run() {

    for (;;) {
        int[] n = new int[cnt];
        // simulate work load
        busy(wcet);
        n = null;
        waitForNextPeriod();
    }
}

final static int MIN_US = 10;

static void busy(int us) {

    int t1, t2, t3;
    int cnt;

    cnt = 0;
    // get the current time in us
    t1 = Native.rd(Const.IO_US_CNT);

    for (;;) {
        t2 = Native.rd(Const.IO_US_CNT);
        t3 = t2-t1;
        t1 = t2;
        if (t3<MIN_US) {
            cnt += t3;
        }
        if (cnt>=us) {
            return;
        }
    }
}
```

For the busy loop to simulate *real* execution time, and not elapsed time, the constant MIN_US has to be less than the time for two context switches, but larger than the execution time of one iteration of the busy loop. In this case only cycles executed by the busy loop are counted for the execution time and interruption due to a higher priority thread is not part of the execution time measurement.

In our example we use a concurrent-copy collector with a heap size (for both semi-spaces) of 100 KB. At startup the JVM allocates about 3.5 KB data. We incorporate[11] these 3.5 KB as static live data L_s.

The remaining memory at the end of the GC cycle, shown in the following graphs, give an indication of the conservatism of the GC period bound. The assumed worst-case for the GC period would result in almost zero free memory before the semi-space flip.

---

[11] The suggested handling of static data to be moved to *immortal* memory at mission start is not yet implemented.

	T_i	C_i	a_i
τ_1	5 ms	1 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_{GC}	77 ms	11 ms	

Table 1 Thread properties for experiment 1

4.2.1 Independent Threads

The first example consists of two threads with the properties listed in Table 1. T_i is the period, C_i the WCET, and a_i the maximum amount of memory allocated each period. Note that the period for the collector thread is also listed in the table although it is a result of the worker thread properties and the heap size.

With the periods T_i and the memory consumption a_i for the two worker threads we calculate the maximum period T_{GC} for the collector thread τ_{GC} by using Theorem 1

$$
\begin{aligned}
T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \\
&\leq \frac{100 - 2(3.5 + 4) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10}\right)} \text{ms} \\
&\leq 77 \text{ms}
\end{aligned}
$$

The priorities are assigned rate-monotonic [19] and we perform a quick schedulability check with the periods T_i and the WCETs C_i by calculation of the processor utilization U for all three threads

$$
\begin{aligned}
U &= \sum_{i=1}^{3} \left(\frac{C_i}{T_i}\right) \\
&= \frac{1}{5} + \frac{3}{10} + \frac{11}{77} \\
&= 0.643
\end{aligned}
$$

which is less than the maximum utilization for three tasks

$$
\begin{aligned}
U_{max} &= m * \left(2^{\frac{1}{m}} - 1\right) \\
&= 3 * \left(2^{\frac{1}{3}} - 1\right) \\
&\approx 0.78
\end{aligned}
$$

In Figure 4 the memory trace for this system is shown. The graph shows the free memory in one semi-space (the to-space, which is 50 KB) during the execution of the application. The individual points are recorded with time-stamps at the end of each allocation request.

In the first milliseconds we see allocation requests that are part of the JVM startup (most of it is static data). The change to the mission phase is delayed 100 ms and the first allocation from a periodic thread is at 105 ms. The collector thread also starts at
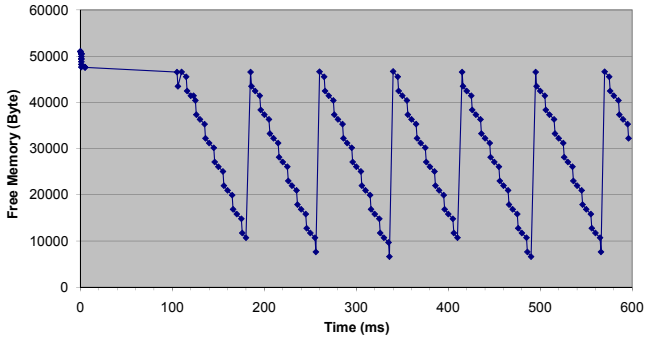
Fig. 4 Free memory in experiment 1

	T_i	C_i	a_i
τ_1	5 ms	0.5 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_3	30 ms	2 ms	
τ_{GC}	55 ms	12 ms	

Table 2 Thread properties for experiment 2

the same time and the first semi-space flip can be seen at 110 ms (after one allocation from each worker thread). We see the 77 ms period of the collector in the jumps in the free memory graph after the flip. The different memory requests of two times 1 KB from thread τ_1 and one time 3 KB from thread τ_2 can be seen every 10 ms.

In this example the heap is used until it is almost full, but the application never runs out of memory and no thread misses a deadline. From the regular allocation pattern we also see that this collector runs concurrently. With a stop-the-world collector we would notice gaps of 10 ms (the measured execution time of the collector) in the graph.

4.2.2 Producer/Consumer Threads

For the second experiment we split our thread τ_1 to a producer thread τ_1 and a consumer thread τ_3 with a period of 30 ms. We assume after the split that the producer's WCET is halved to 500 us. The consumer thread is assumed to be more efficient when working on lager blocks of data than in the former example (C_3=2 ms instead of 6*500 μs). The rest of the setting remains the same (the worker thread τ_2). Table 2 shows the thread properties for the second experiment.

As explained in Section 2.3.3, we calculate the lifetime factor l_1 for memory allocated by the producer τ_1 with the corresponding consumer τ_3 with period T_3.

$$l_1 = \left\lceil \frac{2T_3}{T_1} \right\rceil = \left\lceil \frac{2 \times 30}{5} \right\rceil = 12$$
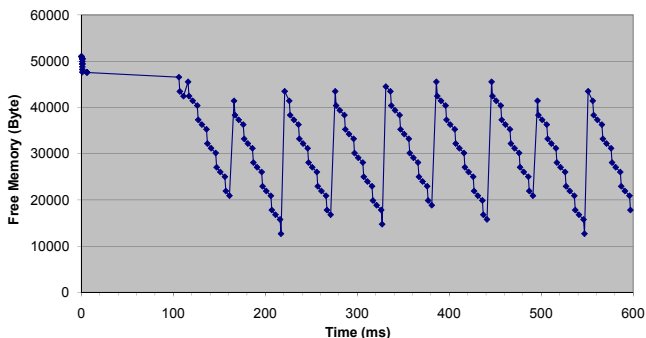
Fig. 5 Free memory in experiment 2

The maximum collector period T_{GC} is

$$
\begin{aligned}
T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i l_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}} \\
&\leq \frac{100 - 2(3.5 + 1 \cdot 12 + 3 + 0) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10} + \frac{0}{30}\right)} \mathrm{ms} \\
&\leq 55\mathrm{ms}
\end{aligned}
$$

We check the maximum processor utilization:

$$
\begin{aligned}
U &= \sum_{i=1}^{4} \left(\frac{C_i}{T_i}\right) \\
&= \frac{0.5}{5} + \frac{3}{10} + \frac{2}{30} + \frac{12}{55} \\
&= 0.685 \leq 4 * (2^{\frac{1}{4}} - 1) \approx 0.76
\end{aligned}
$$

In Figure 5 the memory trace for the system with one producer, one consumer, and one independent thread is shown. Again, we see the 100 ms delayed mission start after the startup and initialization phase, in this example at about 106 ms. Similar to the former example the first collector cycle performs the flip a few milliseconds after the mission start. We see the shorter collection period of 55 ms. The allocation pattern (two times 1 KB and one time 3 KB per 10 ms) is the same as in the former example as the threads that allocate the memory are still the same.

We have also run this experiment for a longer time than shown in Figure 5 to see if we find a point in the execution trace where the remaining free memory is less than the value at 217 ms. The pattern repeats and the observed value at 217 ms is the minimum.

Listing 5 Measuring release time jitter

```
public boolean run() {

    int t = Native.rdMem(Const.IO_US_CNT);
    if (!notFirst) {
        expected = t+period;
        notFirst = true;
    } else {
        int diff = t-expected;
        if (diff>max) max = diff;
        if (diff<min) min = diff;
        expected += period;
    }
    work();

    return true;
}
```

4.3 Measuring Release Jitter

Our main concern on garbage collection in real-time systems is the blocking time introduced by the GC due to atomic code sections. The blocking time cannot be measured directly. However, it will be seen as release time jitter on the high priority real-time threads. Therefore we measure this release jitter.

Listing 5 shows how we measure the jitter. Method run() is the main method of the real-time thread and executed on each periodic release. Within the real-time thread we have no notion about the start time of the thread. As a solution we measure the actual time on the first iteration and use this time as first release time. In each iteration the expected time, stored in the variable expected, is incremented by the period. In each iteration (except the first one) the actual time is compared with the expected time and the maximum value of the difference is recorded.

As noted before, we have no notion about the *correct* release times. We measure only relative to the first release. When the first release is delayed (due to some startup code or interference with a higher priority thread) we have a positive offset in expected. On an exact release in a later iteration the time difference will be negative (in diff). Therefore, we also record the minimum value for the difference between the actual time and the expected time. The maximum measured release jitter is the difference between max and min.

To provide a baseline we measure the release time jitter of a single real-time thread (plus an endless loop in the main method as an idle non-real-time background thread). No GC thread is scheduled. The code is similar to the code in Listing 5. A stop condition is inserted that prints out the minimum and maximum time differences measured after 1 million iterations.

Table 3 shows the measured jitter for different thread periods. We observed no jitter for periods of 100 μs and longer. At a period of 50 μs the scheduler introduces a considerable amount of jitter. From this measurement we conclude that 100 μs is

Period	Jitter
200 μs	0 μs
100 μs	0 μs
50 μs	17 μs

Table 3 Maximum release jitter for a single thread

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	5
τ_p	1 ms	1 ms	4
τ_c	10 ms	10 ms	3
τ_{log}	1000 ms	100 ms	2
τ_{gc}	200 ms	200 ms	1

Table 4 Thread properties of the test program

the practical shortest period we can handle with our system. We will use this period for the high-priority real-time thread in the following measurement with an enabled GC.

4.3.1 Measurements

The test application consisting of three real-time threads (τ_{hf}, τ_p, and τ_c), one logging thread τ_{log}, and the GC thread τ_{gc}. All three real-time threads measure the difference between the expected release time and the actual release time (as shown in Figure 5). The minimum and maximum values are recorded and regularly printed to the console by the logging thread τ_{log}. Table 4 shows the release parameters for the five threads. Priority is assigned deadline monotonic. Note that the GC thread has a shorter period than the logger thread, but a longer deadline. For our approach to work correctly the GC thread *must* have the lowest priority. Therefore all other threads with a longer period than the GC thread must be assigned a shorter deadline.

Thread τ_{hf} represents a high-frequency thread without dynamic memory allocation. This thread should observe minimal disturbance by the GC thread.

The threads τ_p and τ_c represent a producer/consumer pair that uses dynamically allocated memory for communication. The producer appends the data at a frequency of 1 kHz to a simple list. The consumer thread runs at 100 Hz and processes all currently available data in the list and removes them from the list. The consumer will process between 9 and 11 elements (depending on the execution time of the consumer and the thread phasing).

It has to be noted that this simple and common communication pattern cannot be implemented with the scoped memory model of the RTSJ. First, to use a scope for communication, we have to keep the scope alive with a *wedge* thread [23] when data is added by the producer. We would need to notify this wedge thread by the consumer when all data is consumed. However, there is no single instant available where we can *guarantee* that the list is empty. A possible solution for this problem is described in [23] as *handoff* pattern. The pattern is similar to double buffering, but with an explicit

Threads	Jitter
τ_{hf}	$0\ \mu s$
τ_{hf}, τ_{log}	$7\ \mu s$
$\tau_{hf}, \tau_{log}, \tau_p, \tau_c$	$14\ \mu s$
$\tau_{hf}, \tau_{log}, \tau_p, \tau_c, \tau_{gc}$	$54\ \mu s$

Table 5 Jitter measured on a 100 MHz processor for the high priority thread in different configurations

copy of the data. The elegance of a simple list as buffer queue between the producer and the consumer is lost.

Thread τ_{log} is not part of the real-time systems simulated application code. Its purpose is to print the minimum and maximum differences between the measured and expected release times (see former section) of threads τ_{hf} and τ_p to the console periodically.

Thread τ_{gc} is a standard periodic real-time thread executing the GC logic. The GC thread period was chosen quite short in that example. A period in the range of seconds would be enough for the memory allocation by τ_p. However, to stress the interference between the GC thread and the application threads we artificially shortened the period.

As a first experiment we run only τ_{hf} and the logging thread τ_{log} to measure jitter introduced by the scheduler. The maximum jitter observed for τ_{hf} is 7 μs – the blocking time of the scheduler.

In the second experiment we run all threads except the GC thread. For the first 4 seconds we measure a maximum jitter of 14 μs for thread τ_{hf}. After those 4 seconds the heap is full and GC is necessary. In that case the GC behaves in a stop-the-world fashion. When a new object request cannot be fulfilled the GC logic is executed in the context of the allocating thread. As the bytecode new is itself in an atomic region the application is blocked until the GC finishes. Furthermore, the GC performs a conservative scan of all thread stacks. We measure a release delay of 63 ms for all threads due to the blocking during the full collection cycle. From that measurement we can conclude for the sample application and the available main memory: (a) the measured maximum period of the GC thread is in the range of 4 seconds; (b) the estimated execution time for one GC cycle is 63 ms. It has to be noted that measurement is not a substitution for static timing analysis. Providing WCET estimates for a GC cycle is a challenge for future work.

In our final experiment we enabled all threads. The GC is scheduled periodically at 200 ms as the lowest priority thread – the scenario we argue for. The GC logic is set into the concurrent mode on mission start. In this mode the thread stacks are not scanned for roots. Furthermore, when an allocation request cannot be fulfilled the application is stopped. This radical stop is intended for testing. In a more tolerant implementation either an out-of-memory exception can be thrown or the requesting thread has to be blocked, its thread stack scanned and released when the GC has finished its cycle.

We ran the experiment for several hours and recorded the maximum release jitter of the real-time threads. For this test we used slightly different periods (prime numbers) to avoid the regular phasing of the threads. The harmonic relation of the original

periods can lead to too optimistic measurements. The applications never ran out of memory. The maximum jitter observed for the high priority task τ_{hf} was 54 μs. The maximum jitter for task τ_p was 108 μs. This higher value on τ_p is expected as the execution interferes with the execution of the higher priority task τ_{hf}.

5 Discussion

In this section, we discuss our findings from the evaluation of the proposed GC on the Java processor JOP and related aspects for real-time GC.

5.1 GC Scheduling

The scheduling experiments give as some confidence that the calculation of the maximum period of the GC is correct. The implementation of bytecode new has been instrumented to collect statistics of the heap usage. The time series of the heap usage shows expected patterns. The free memory drops near to zero at the end of the GC period, but jumps up after the flip on the start of the next GC period.

5.2 Allocation Rate Analysis

To bound the maximum GC period, the allocation rate of the individual mutator threads needs to be known. This problem is similar to schedulability analysis where the WCET of individual tasks needs to be known. The maximum allocation rate analysis is very similar to WCET analysis. Instead of execution time of individual instructions, the allocation size is used as cost function in the analysis. Besides bounds on loops and recursion depths, a bound on array sizes is needed for this analysis. The WCET analysis tool for Java processors [34] is currently adapted to perform a maximum allocation rate analysis. This type of analysis is also necessary for RTSJ style scoped memories.

The analysis of memory consumption by objects that are shared between tasks for communication is more complex. An inter-task analysis is necessary to derive an upper bound on the time an object stays alive. For simple consumer/producer task pairs the analysis can be done manually. Automatic analysis of complex object sharing is a challenging future work project.

5.3 Blocking Times

With our measurements we have shown that quite short blocking times are achievable. Scheduling introduces a blocking time of about 7–14 μs and the GC adds another 40 μs resulting in a maximum jitter of the highest priority thread of 54 μs. In our first implementation we performed the object copy in pure Java, resulting in blocking times around 200 μs. To speedup the copy we moved this function to microcode.

However, the microcoded *memcpy* still needs 18 cycles per 32-bit word copy. Direct support of object copy in hardware completely removes the blocking time [33].

The maximum blocking time of 54 μs on a 100 MHz processor is less than blocking times reported for other solutions.

Blocking time for Metronome (called pause times in the papers) is reported to be 6 ms [6] on a 500 MHz PowerPC at 50% CPU utilization. Those large blocking times are due to the scheduling of the GC at the highest priority with a polling based yield within the GC thread. A fairer comparison is against the *jitter* of the pause time. In [5] the variation of the pause time is given between 500 μs and 2.4 ms on a 1 Ghz machine. It should be noted that Metronome is a GC intended for mixed real-time systems whereas we aim only for hard real-time systems.

Robertz performed a similar measurement as we did for his thesis [26] with a time-triggered GC on a 350 MHz PowerPC. He measured a maximum jitter of 20 μs (± 10 μs) for a high priority task with a period of 500 μs.

It has to be noted that our experiment is a small one and we need more advanced real-time applications for the evaluation of real-time GC. The problem is that it is hard to find even static based real-time application benchmarks (at least applications written for safety critical Java). Running standard benchmarks that measure average case performance (e.g., SPEC jvm98) is not an option to evaluate a real-time collector.

5.4 Sizing the Handle Area

The lightweight read barrier with an indirection to find the correct object can either be a word in the object header or located in a distinct handle area. A handle area uses less memory as the indirection word is needed only once and not in both copies of an object. Furthermore, the handle area can also hold other typical object header data, such as a lock field or the array size.

The main drawback of a handle area is that this area needs to be sized. As we already know the maximum allocation rate and live data for each thread the maximum number of handles can be derived by an adaption of the scheduling equations. For thread τ_i that allocates n_i objects each period the maximum number of handles N_{max} is

$$N_{max} \leq \sum_{i=1}^{n} n_i + \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil n_i$$

5.5 Can a Out-of-Memory Error Happen?

From a hard real-time point of view the application has to be fully analyzed and an out-of-memory error shall never happen. We could ignore that fault. However, from a safety-critical point of view one should provide a backup solution even for the not-expected case. Instead of throwing an OutOfMemoryError in the allocating thread, the thread can be blocked until the GC frees enough memory (till the flip in the new GC cycle). The solution needs some cooperation between the GC code and the scheduler.

5.6 Mixed Real-Time Systems

Most implementations of a real-time GC within the RTGC target soft real-time or mixed real-time systems. The hard real-time part shall use scoped memory and no-heap real-time threads and the soft realtime part of the application shall use real-time threads using the heap.

Another option to split the application between hard and soft real-time components is to split the garbage collected heap. To protect hard real-time threads they allocate in a reserved part of the heap. The analysis from Section 2 only needs to be applied for the hard real-time part. The soft (or non) real-time components are allowed to allocate in a different part of the heap. If this part of the heap becomes full a soft real-time thread will block on allocation till the GC frees enough memory. The hard real-time threads are not influenced.

It has to be noted that data sharing between the two components is still allowed. As both heap regions are under the control of a single GC thread, pointers between the two regions are allowed and no further treatment of cross references is needed.

6 Related Work

Garbage collection was first introduced for list processing systems (LISP) in the 1960s. A good overview of GC techniques can be found in [17] and in the GC survey by Wilson [39].

Mark-sweep and mark-compact collectors need a stack during the marking phase that can grow in the worst-case up to the number of live objects. Cheney [13] presents an elegant way how this mark stack can be avoided. His GC is called *copying-collector* and divides the heap into two spaces: the *to-space* and the *from-space*. Objects are moved from one space to the other as part of the scan of the object graph.

However, all the described collectors are still stop-the-world collectors. The pause time of up to seconds in large interactive LISP applications triggered the research on incremental collectors that distribute collection work more evenly [38,14,8]. These collectors were sometimes called *real-time* although they do not fulfill hard real-time properties that we need today. Baker [8] extends Cheneys [13] copying collector for incremental GC. However, it uses an expensive read barrier that moves the object to the to-space as part of the mutator work. Baker proposes the *Treadmill* [9] to avoid copying. However, this collector works only with objects of equal size and still needs an expensive read barrier.

In [27] a garbage-collected memory module is suggested to provide a real-time collector. A worst-case delay time of $1\mu s$ is claimed without giving the processor speed.

Metronome is a collector intended for mixed real-time systems [7]. Non real-time applications are used (SPECjvm98) in the experiments. They propose a collector with constant utilization to meet real-time requirements. However, utilization is *not* a real-time measure per se; it should be schedulability or response time instead. In contrast to our proposal, the GC thread is scheduled at the highest priority in short periods. To ensure that, despite the high priority of the GC thread, mutator threads will be

scheduled, the GC thread runs only for a fraction of time within a time window. This fraction and the size of the time window can be adjusted for different work loads.

A comparison of the schedulability of a GC according to this paper and the Metronome GC can be found in [18]. The paper provides response time analysis for a Metronome style GC and shows that it can be used for hard real-time systems. Two constructed example workloads are given. In one case the Metronome based GC will run out of memory and in the other case the GC scheduled at lowest priority will run out of memory. However, the second example has a task with a longer period than the GC thread. According to our proposal the GC thread should then be scheduled with a higher priority, which is not considered in the paper. Experimental comparison of the two GC approaches shows that the response time of the real-time tasks is shorter when the GC thread is scheduled as proposed in this paper.

Although not mandated, all commercial and academic implementations of the RTSJ [37, 11, 4, 2] and related real-time Java systems [1] also contain a real-time garbage collector.

The work closest to our scheduling analysis is presented in [25]. The authors provide an upper bound of the GC cycle. Although stated that this bound "is thus not dependent of any particular GC algorithm", the result applies only for single heap GC algorithms (e.g. mark-compact) and not for a copying collector. A value for L_{max} is not given in the paper. Furthermore, the increase of the object lifetime due to object sharing between threads is not considered in that paper.

7 Conclusion

In this paper we have presented a real-time garbage collector in order to benefit from a more dynamic programming model for real-time applications. The collector is incremental and scheduled as a normal real-time thread and assigned a priority according to its deadline. To guarantee that the applications will not run out of memory, the period of the collector thread has to be short enough. We provided the maximum collector periods for a mark-compact collector type and a concurrent-copy collector. We have also shown how a longer lifetime due to object sharing between threads can be incorporated into the collector period analysis.

The restrictions from the safety-critical Java programming model and the low priority collector thread result in two advantages: (a) avoidance of stack root scanning and (b) short blocking times of high priority threads. At 100 MHz we measured 40 μs maximum blocking time introduced by the GC thread.

A critical operation for a concurrent, compacting GC is the atomic copy of large arrays. JOP has been extended by a copy unit that can be interrupted. This unit is integrated with the memory access unit and redirects the access to either fromspace or tospace depending on the array/field index and the value of the copy pointer.

References

1. Aonix. Perc pico 1.1 user manual. http://research.aonix.com/jsc/pico-manual.4-19-08.pdf, April 2008.

2. Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

3. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

4. Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 249–258, New York, NY, USA, 2007. ACM.

5. David F. Bacon, Perry Cheng, David Grove, Michael Hind, V. T. Rajan, Eran Yahav, Matthias Hauswirth, Christoph M. Kirsch, Daniel Spoonhower, and Martin T. Vechev. High-level real-time programming in java. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 68–78, New York, NY, USA, 2005. ACM Press.

6. David F. Bacon, Perry Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In Robert Meersman and Zahir Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478. Springer, 2003.

7. 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 Press.

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

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

10. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

11. Gregory Bollella, Bertrand Delsart, Romain Guider, Christophe Lizzi, and Frédéric Parain. Mackinac: Making HotSpot^{TM} real-time. In *ISORC*, pages 45–54. IEEE Computer Society, 2005.

12. Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Prgrm. Chrm. G. L. Steele, Jr., editor, *LISP and Functional Programming. Conference Record of the 1984 ACM Symposium, Austin, Texas, August 6-8, 1984*, number ISBN 0-89791-142-3, New York, 1984. ACM.

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

14. 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.

15. Flavius Gruian and Zoran Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.

16. Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, York, United Kingdom, Mar. 2009.

17. Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

18. Tomás Kalibera, Filip Pizlo, Antony L. Hosking, and Jan Vitek. Scheduling hard real-time garbage collection. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 81–92. IEEE Computer Society, 2009.

19. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

20. Albert F. Niessner and Edward G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS*, 2003.

21. S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 113–133. Springer-Verlag, 1987. Lecture Notes in Computer Science 274; Proceedings of Conference held at Portland, OR.

22. Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 77–84, New York, NY, USA, 2006. ACM Press.

23. Filip Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on, Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110, 2004.

24. Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 68–76, Santa Clara, California, September 2008.

25. Sven Gestegard Robertz and Roger 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 Press.

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

27. William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

28. Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

29. Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.

30. Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 57–62, Vienna, Austria, September 2007. ACM Press.

31. Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

32. Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

33. Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 77–84, Santa Clara, California, September 2008. ACM Press.

34. Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Software: Practice and Experience*, 40/6:507–542, 2010.

35. Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.

36. Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

37. Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

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

39. Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

40. Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.