Real-Time Garbage Collection for Java

Schoeberl, Martin

Published in:
Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)

Link to article, DOI:
10.1109/ISORC.2006.66

Publication date:
2006

Document Version
Early version, also known as pre-print

Citation (APA):
Abstract

Automatic memory management or garbage collection greatly simplifies the 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 concurrent collector that is scheduled periodically in the same way as ordinary application threads. We provide an upper bound for the collector period so that the application threads never run out of memory.

1. Introduction

Garbage Collection (GC) is an essential part of the Java runtime system. GC enables automatic dynamic memory management which is essential to build large applications. Automatic memory management frees the programmer from complex and error prone explicit memory management. However, garbage collection is considered unsuitable for real-time systems due to the unpredictable blocking times introduced by the GC work. As one solution to use Java for real-time systems the Real-Time Specification for Java (RTSJ) [4] introduces new thread types with program managed, scoped memory for dynamic memory requirements. This scoped memory (and static memory called immortal memory) is not managed by the GC.

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 be incremental with a short maximum blocking time that has to be known
- The GC has to keep up with the garbage generated by the application threads to avoid out-of-memory stalls

The first point is necessary to limit interference between the GC thread and high-priority threads. The second 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. We provide a solution to calculate the maximum period of the GC thread that collects enough memory in each collector cycle so we never run out of memory. The collector cycle depends on the heap size and the allocation rate of the application threads.

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 bounded memory allocation rate with minimal temporal interference to tasks that use only static memory.

The collector presented in this paper is based on [13, 6, 2]. However, the copying collector performs the copy of an object concurrent by the collector and not as part of the mutator work. Therefore we name it concurrent-copy collector. We use the terms first introduced in [6]. 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 simple called collector.

The paper is structured as follows: Section 2 provides an introductory example of concurrent collection for a mark-compact and a copy collector. In Section 3 the minimum heap size for a mark-compact and copying collector are given. It is shown that the necessary heap size for a mark-compact collector is similar to the heap size for a copying collector. Based on the findings from Section 3 we provide an upper bound for the collector period so that the application threads never run out of memory in Section 4. It is also shown how producer/consumer threads and static objects can be incorporated in the collector period analysis.
Furthermore we provide a simpler solution for static objects than the immortal memory in the RTSJ. In Section 5 we show experiments where the results from this paper are applied on the first prototype implementation of a concurrent-copy collector on a Java processor. Section 6 presents related work and the paper is concluded in Section 7.

This paper makes a strong statement to consider a periodic scheduled, concurrent garbage collector as an option for hard real-time systems. The collector period, besides the WCET of the collector, is the single parameter of the collector that can be incorporated in standard schedulability analysis.

2. An Example

We start our discussion with a simple example\(^1\) where the collector period is 3 times the mutator period and a heap size of 8 objects. 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: \(g\) is garbage from mutator cycle \(i\), \(l\) 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 there are three objects \((g_1, g_2, \text{ and } g_3)\) 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 the second sub-figure). At the end of the cycle, just before compacting, we have three garbage cells \((g_1\text{-}g_3)\), three floating garbage cells \((f_4\text{-}f_6)\) and one live cell \(l_7\). Compaction moves the floating garbage and the live cell to the start of the heap and we end up with four free cells. 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 the collection cycle with three garbage cells left over from the last cycle and one live cell. 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. The second sub-figure shows a snapshot of the heap during the collection. Former 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 the from-space contains the three garbage cells \((g_1\text{-}g_3)\) and the to-space the three floating garbage cells \((f_4\text{-}f_6)\) and one live cell \(l_7\). The last sub-figure 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).

---

\(^1\)The relation between the heap size and the mutator/collector proportion is an arbitrary value in this example. We provide the exact values in the next sections.
3. 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.

The following symbols are used for the rest of the paper:\n\begin{align*}
\text{heap size for a mark-compact collector } (H_{\text{MC}}) & \quad \text{and for a concurrent-copying collector } (H_{\text{CC}}) \text{ containing both semi-spaces, period of the GC thread } (T_{\text{GC}}) \text{, period of a single mutator thread } (T_M) \text{, period of mutator thread } i \ (T_i) \text{ from a set of threads, and memory amount allocated by a single mutator } (a) \text{ or by mutator } i \ (a_i) \text{ from a set of threads.}
\end{align*}

3.1. Mark-Compact

For the mark-compact collector the heap $H_{\text{MC}}$ can be divided into allocated memory $M$ and free memory $F$
\begin{equation}
H_{\text{MC}} = M + F = G + \overline{G} + L + F
\end{equation}
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 the heap size $H_{\text{MC}}$
\begin{equation}
H_{\text{MC}} \geq M = G + \overline{G} + L
\end{equation}

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 $L_{\text{max}}$ at the start of the collection cycle and two times $A_{\text{max}}$, the maximum data allocated by the mutator during the collection cycle.
\begin{equation}
M \leq L_{\text{max}} + 2A_{\text{max}}
\end{equation}

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 $n + 1$.
\begin{equation}
G^{n+1} = \overline{G}^n
\end{equation}
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.
\begin{equation}
L_S + A = L_E + \overline{G}_E
\end{equation}
with $A \leq A_{\text{max}}$ and $L_S \leq L_{\text{max}}$. A new collection-cycle start immediately follows the end of the former cycle. Therefore the live data remains unchanged.
\begin{equation}
L_{n+1}^S = L_n^S
\end{equation}

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.
\begin{equation}
M_1^S = 0
\end{equation}

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.
\begin{equation}
A^1 = L_1^E + \overline{G}_1^E
\end{equation}

Therefore at the end of the first cycle
\begin{align*}
M_1^1 & = L_1^E + \overline{G}_1^E
M_1^1 & = A^1
\end{align*}

As $A^1 \leq A_{\text{max}}$ (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 $n + 1$.
\begin{equation}
M^{n+1} \leq L_{\text{max}} + 2A_{\text{max}}
\end{equation}

\begin{align*}
M^n & = G^n + \overline{G}_E^n + L^n_S
M^{n+1} & = G^{n+1} + \overline{G}^{n+1}_E + L^{n+1}_S
& = \overline{G}_E^n + L^{n+1}_S + A^{n+1} \quad \text{apply (4) and (5)}
& = \overline{G}_E^n + L^n_E + A^{n+1} \quad \text{apply (6)}
& = L^n_S + A^n + A^{n+1} \quad \text{apply (5)}
\end{align*}

As $L_S \leq L_{\text{max}}$, $A^n \leq A_{\text{max}}$ and $A^{n+1} \leq A_{\text{max}}$
\begin{equation}
M^{n+1} \leq L_{\text{max}} + 2A_{\text{max}}
\end{equation}

\section{3.2. Concurrent-Copy}

In the following we denote the maximum allocated memory in the from-space as $M_{\text{from}}$ and the maximum allocated memory in the to-space as $M_{\text{to}}$.
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 $\frac{H_{CC}}{2}$

$$H_{CC} = H_{From} + H_{To} \geq 2M$$  \hspace{1cm} (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.

$$M \leq L_{\max} + A_{\max}$$  \hspace{1cm} (16)

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

$$G_{n+1} = \overline{G}_{E}$$  \hspace{1cm} (17)

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

$$L_{S}^{n+1} = L_{E}^{n}$$  \hspace{1cm} (18)

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

$$M_{From} = G + L_S$$  \hspace{1cm} (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.

$$M_{To} = \overline{G} + L$$  \hspace{1cm} (20)

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

$$M_{To,S} = 0$$  \hspace{1cm} (21)

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

$$M_{To,E} = L_S + A$$  \hspace{1cm} (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 $\overline{G}_{E}$.

$$L_S + A = L_E + \overline{G}_{E}$$  \hspace{1cm} (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 ($M_{From} = 0$). 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 $n + 1$.

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

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

$$H_{From}^{n+1} \Leftrightarrow H_{To}^{n}$$
$$H_{To}^{n+1} \Leftrightarrow \emptyset$$  \hspace{1cm} (25)

The from-space:

$$M_{From}^{n} = G^n + L_{S}^{n}$$  \hspace{1cm} (26)
$$M_{From}^{n+1} = G_{n+1}^{n} + L_{S}^{n+1}$$  \hspace{1cm} (27)

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

$$M_{From}^{n+1} \leq L_{\max} + A_{\max}$$  \hspace{1cm} (29)

The to-space:

$$M_{To}^{n} = \overline{G}_{E}^{n} + L_{E}^{n}$$  \hspace{1cm} (30)
$$M_{To}^{n+1} = \overline{G}_{E}^{n+1} + L_{E}^{n+1}$$  \hspace{1cm} (31)

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

$$M_{To}^{n+1} \leq L_{\max} + A_{\max}$$  \hspace{1cm} (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 need double of 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 do not need to be copied. The compacting collector moves all newly created data (that is mostly floating garbage) at the compaction phase.

### 4. Garbage Collection Period

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.

### 4.1. 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

$$T_{GC} \leq T_M \left\lceil \frac{H_{MC} - a}{2a} \right\rceil$$ (34)

$$T_{GC} \leq T_M \left\lceil \frac{H_{CC} - 2a}{2a} \right\rceil$$ (35)

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

$$L_{max} = a$$ (36)

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

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

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

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

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

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

$$H_{CC} \geq 2(L_{max} + A_{max})$$

$$H_{CC} \geq 2a \left(1 + \frac{T_{GC}}{T_M}\right)$$ (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 $H_{CC}$

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

The maximum quotient $\left\lceil \frac{T_{GC}}{T_M} \right\rceil$ 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

$$\frac{T_{GC}}{T_M} \leq \left\lceil \frac{H_{MC} - a}{2a} \right\rceil$$ (42)

$$\Rightarrow T_{GC} \leq T_M \left\lceil \frac{H_{MC} - a}{2a} \right\rceil$$ (43)

and for the concurrent-copy collector

$$\frac{T_{GC}}{T_M} \leq \left\lceil \frac{H_{CC} - 2a}{2a} \right\rceil$$ (44)

$$\Rightarrow T_{GC} \leq T_M \left\lceil \frac{H_{CC} - 2a}{2a} \right\rceil$$ (45)

### 4.2. 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 several ($n$) 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

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

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

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

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

$$A_{max}' = \sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1\right) a_i \geq \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i$$ (50)
From (2) and (3) we get the minimum heap size for a mark-compact collector

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

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

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

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

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

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

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

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

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

$$\geq 2\sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n} \left[\frac{T_{GC}}{T_i}\right] a_i \tag{55}$$

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

$$2A_{max}' \leq H_{CC} - 2L_{max}$$

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

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

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

4.3. 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 $\tau_p$ and $T_c$ as the period of the consumer thread $\tau_c$. $\tau_p$ allocates $a_p$ memory each period. During one period of the consumer $\tau_c$ the producer $\tau_p$ allocates

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

memory. The worst case is that $\tau_c$ 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

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

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

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

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

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

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

A mark-compact collector moves all static data to the bottom of the heap in the first 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 would move these static data in each collection cycle. Furthermore, the memory demand for the copy collector is increased by the double amount of the static data (compared to the single amount in the mark-compact collector)\(^2\).

As these static objects live forever, we propose a similar solution to the immortal memory of the RTSJ. We divide our application into an initialization and a mission phase [10]. All static data is allocated during the initialization phase (where no application threads are scheduled). As part of the transition to the mission phase we perform a special collection cycle in a stop-the-world fashion. Live data that exists after this cycle are assumed to be immortal data and make up the immortal memory area. The remaining memory is used for the garbage collected heap.

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

- 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 $L_{\text{max}}$ only contains allocated memory from the periodic threads.

5. Experiments

In this section we test an implementation of the concurrent-copy garbage collector on a Java processor (JOP, [12]). 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 10.4ms when executing it with scheduling disabled during one collection cycle for example 1 and 11.2ms for example 2. We use 11ms and 12ms respectively as the WCET of the collector for the following examples\(^3\).

In our example we use a concurrent-copy collector with a heap size (for both semi-spaces) of 100KB. At startup the JVM allocates about 3.5KB data. We incorporate\(^4\) these 3.5KB as static live data $L_s$.

5.1. 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 from the application threads 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 $T_{\text{GC}}$ for the collector thread $T_{\text{GC}}$ by using Theorem 1

$$T_{\text{GC}} \leq \frac{H_{\text{CC}} - 2 (L_s + \sum_{i=1}^{n} a_i) - 2 \sum_{i=1}^{n} a_i}{2 \sum_{i=1}^{n} \frac{a_i}{T_i}}$$

$$\leq \frac{100 - 2 \left(3.5 + 4\right) - 2 \cdot 4}{2 \left(\frac{1}{5} + \frac{3}{10}\right)} \text{ms} = 77\text{ms}$$

The priorities are assigned rate-monotonic [8] 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

$$U = \sum_{i=1}^{3} \left(\frac{C_i}{T_i}\right) = \frac{1}{5} + \frac{3}{10} + \frac{11}{17} = 0.643$$

which is less than the maximum utilization for three tasks

$$U_{\text{max}} = m \ast (2^{\frac{1}{m}} - 1) = 3 \ast (2^{\frac{1}{3}} - 1) \approx 0.78$$

In Figure 3 the memory trace for this system is shown. The graph shows the free memory in one semi-space (the

\(^2\)Or the collector period gets shortened.

\(^3\)It has to be noted that measuring execution time is not a safe method to estimate WCET values.

\(^4\)We have not yet implemented the suggested handling of static data to be moved to immortal memory at mission start in our prototype.
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 100ms and the first allocation from a periodic thread is at 105ms. The collector thread also starts at the same time and the first semi-space flip can be seen at 110ms (after one allocation from each worker thread). We see the 77ms period of the collector in the jumps in the free memory graph after the flip. The different memory requests of two times 1KB from thread $\tau_1$ and one time 3KB from thread $\tau_2$ can be seen every 10ms.

In this example the heap is used until it is almost full, but we run never 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 10ms (the measured execution time of the collector) in the graph.

### 5.2. Producer/Consumer Threads

For the second experiment we split our thread $\tau_1$ to a producer thread $\tau_1$ and a consumer thread $\tau_3$ with a period of 30ms. We assume after the split that the producer’s WCET is halved to 500\(\mu\)s. The consumer thread is assumed to be more efficient when working on larger blocks of data than in the former example ($C_3=2\text{ms}$ instead of 6.500\(\mu\)s). The rest of the setting remains the same (the worker thread $\tau_2$). Table 2 shows the thread properties for the second experiment.

As explained in Section 4.3 we calculate the lifetime factor $l_1$ for memory allocated by the producer $\tau_1$ with the corresponding consumer $\tau_3$ with period $T_3$.

\[ l_1 = 2 \left\lceil \frac{T_3}{T_1} \right\rceil = 2 \left\lceil \frac{30}{5} \right\rceil = 12 \]

The maximum collector period $T_{GC}$ is

\[ T_{GC} \leq \frac{H_{CC} - 2(L_s + \sum_{i=1}^{n} a_i l_i) - 2 \sum_{i=1}^{n} a_i}{2 \sum_{i=1}^{n} a_i T_i} \]

\[ \leq 100 - 2(3.5 + 1 \cdot 12 + 3 + 0) - 2 \cdot 4 \text{ms} = 55\text{ms} \]

We check the maximum processor utilization:

\[ U = \sum_{i=1}^{4} \left( \frac{C_i}{T_i} \right) = \frac{0.5}{5} + \frac{3}{10} + \frac{2}{30} + \frac{12}{55} \]

\[ = 0.685 \leq 4 \cdot (2^4 - 1) \approx 0.76 \]

In Figure 4 the memory trace for the system with one producer, one consumer, and one independent thread is shown. Again, we see the 100ms delayed mission start after the startup and initialization phase, in this example at about 106ms. Similar to the former example the first collector cycle performs the flip a few milliseconds after the mission start.

![Figure 3. Free memory in experiment 1](image1)

![Figure 4. Free memory in experiment 2](image2)

**Table 1. Thread properties for experiment 1**

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$T_i$</th>
<th>$C_i$</th>
<th>$a_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>5ms</td>
<td>1ms</td>
<td>1KB</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>10ms</td>
<td>3ms</td>
<td>3KB</td>
</tr>
<tr>
<td>$\tau_{GC}$</td>
<td>77ms</td>
<td>11ms</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2. Thread properties for experiment 2**

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$T_i$</th>
<th>$C_i$</th>
<th>$a_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>5ms</td>
<td>0.5ms</td>
<td>1KB</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>10ms</td>
<td>3ms</td>
<td>3KB</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>30ms</td>
<td>2ms</td>
<td></td>
</tr>
<tr>
<td>$\tau_{GC}$</td>
<td>55ms</td>
<td>12ms</td>
<td></td>
</tr>
</tbody>
</table>
start. We see the shorter collection period of 55ms. The allocation pattern (two times 1KB and one time 3KB per 10ms) 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 4 to see if we find a point in the execution trace where the remaining free memory is less than the value at 217ms. The pattern repeats and the observed value at 217ms is the minimum.

6. Related Work

Baker [2] extends Cheneys [5] 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 [3] to avoid copying. However, this collector works only with objects of equal size and still needs an expensive read barrier.

Despite the title [1] is still not a real-time collector. 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. Pause times are in the range of 12ms. In [9] two collectors based on [6] and [3] are implemented on a multi-threaded Microcontroller. Higuera suggests in [7] the use of hardware features from picolJava to speed up RTSJ memory region protection and garbage collection.

In [11] the authors provide an upper bound in the GC cycle time as

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

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. If we use our value of $L_{\max} = \sum_{i=1}^{n} a_i$ the result is the same as in our finding (see Theorem 1) for the mark-compact collector. No analysis is given how objects with longer lifetime and static objects can be incorporated.

7. Conclusion

In this paper we suggest an approach to real-time garbage collection in order to benefit from a more dynamic programming model for real-time applications. The proposed collector is incremental and scheduled periodically in the same way as an application thread.

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 proposed concurrent-copy collector was implemented as a prototype on a Java processor. As the next step we will work on an efficient implementation of the collector and supporting hardware for the write-barrier in JOP [12].

References