Non-blocking Root Scanning for Real-Time Garbage Collection

Wolfgang Puffitsch
Institute of Computer Engineering
Vienna University of Technology, Austria
wpuffits@mail.tuwien.ac.at

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

ABSTRACT

Root scanning is a well known source of blocking times due to garbage collection. In this paper, we show that root scanning only needs to be atomic with respect to the thread whose stack is scanned. We propose two solutions to utilize this fact: (a) block only the thread whose stack is scanned, or (b) shift the responsibility for root scanning from the garbage collector to the application threads. The latter solution eliminates blocking due to root scanning completely. Furthermore, we show that a snapshot-at-beginning write barrier is sufficient to ensure the consistency of the root set even if local root sets are scanned independently of each other. The impact of solution (b) on the execution time of a garbage collector is shown for two different variants of the root scanning algorithm. Finally, we evaluate the resulting real-time garbage collector in a real system to confirm our theoretical findings.

Categories and Subject Descriptors

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

General Terms

Theory, Experimentation

Keywords

garbage collection, real-time, root scanning

1. INTRODUCTION

Tracing garbage collectors traverse the object graph to identify the set of reachable objects. The starting point for this tracing is the *root set*, a set of objects which is known to be directly accessible. On the one hand, these are references in global (static in Java) variables, on the other hand these are references that are local to a thread. The latter comprise the references in a thread's runtime stack and thread-local CPU registers. The garbage collector must ensure that its view of the root set is consistent before it can proceed, otherwise objects could be erroneously reclaimed.

JTRES '08 September 24-26, 2008, Santa Clara, California, USA

For stop-the-world garbage collectors, the consistency of the object graph is trivially ensured. Incremental garbage collectors however require the use of *barriers*, which enforce the consistency of marking and the root set [3, 6, 19, 20]. While barriers are an efficient solution for the global root set, they are considered to be too inefficient to keep the local root sets consistent. Even frequent instructions like storing a reference to a local variable would have to be guarded by such a barrier, which would cause a considerable overhead and make it difficult if not impossible to compute tight bounds for the worst case execution time (WCET). The usual solution to this problem is to scan the stacks of all threads in a single atomic step and stall the application threads while doing so.[1] The atomicity entails that the garbage collector may not be preempted while it scans a thread's stack, which in turn causes a considerable release jitter even for high-priority threads.

However, the atomicity is only necessary w. r. t. the thread whose stack is scanned, because a thread can only modify its own stack. If the garbage collector scans a thread's stack, the thread must not execute and atomicity has to be enforced. Other mutator threads are allowed to preempt the stack scanning thread. If a thread scans its own stack, it is not necessary to prohibit the preemption of the thread – when the thread continues to execute, the stack is still in the same state and the thread can proceed with the scanning without special action. Consequently, preemption latencies due to root scanning can be avoided. With such a strategy, it is also possible to minimize the overhead for root scanning. It can be scheduled in advance such that the local root set is small at the time of scanning.

In this paper, we show the feasibility of this approach. We present two solutions for periodic and sporadic threads, and evaluate their trade-offs. Furthermore, we show how the worst case time until all threads have scanned their stacks can be computed.

We consider a garbage collection approach that uses a separate thread for the garbage collector, as suggested by Henriksson [10]. Such an approach is also used by Robertz [12] and in IBM's Metronome garbage collector [2] to improve timely determinism of garbage collection. For our experiments, we used a concurrent-copy garbage collector as described in [15] and [17]. Our theoretical findings are however not specific to any particular tracing garbage collection algorithm.

The contributions of this paper are:

- Stack scanning performed by the garbage collection thread needs only be atomic w. r. t. to the thread whose stack is scanned.
- Delegating the stack scanning to the mutator threads completely avoids blocking time due to root scanning.

[1] The former implies the latter on uniprocessors, but not on multi-processors.

- A snapshot-at-beginning write barrier [20] is sufficient to protect against losing an object when a reference migrates from a not-yet-scanned stack to a scanned stack.

The paper is organized as follows: In the following section, we describe related work on root scanning in real-time garbage collectors. In Section 3 we present our approach to root scanning and establish its correctness. An implementation of the approach is evaluated in Section 4. Section 5 concludes the paper and presents an outlook on future work.

2. RELATED WORK

The idea of delegating local root scans to the mutator threads is proposed by Doligez, Leroy and Gonthier in [8] and [7]. They point out that this allows for more efficient code and reduces the disruptiveness of garbage collection. Mutator threads should check an appropriate flag from time to time and then scan their local root set. However, the authors remain vague on when the mutators should check this flag and do not investigate the effect of various choices. As they aim for efficiency rather than real-time properties, they do not consider a thread model with known periods and deadlines.

The approach presented in this paper builds to some degree on an approach by Schoeberl and Vitek [17]. They propose a thread model which does not support blocking and where threads cannot retain a local state across periods. They also propose that the garbage collector runs at the lowest priority, which entails that the stacks of all threads are empty when a garbage collection cycle starts. Consequently, the garbage collector only needs to consider the global root set. Although the initial motivations differ, the work presented in this paper can be regarded as generalization of this earlier approach, especially w. r. t. the thread model.

In the JamaicaVM's garbage collector, the mutator threads are also responsible of keeping their root set up to date [18]. However, the JamaicaVM's garbage collector employs a work-based approach, which means that the garbage collector does not execute in a separate thread – garbage collection is distributed among the mutator threads. Therefore, considerations on the correctness and execution time are hardly comparable. In [18], the average overhead for root scanning is estimated as 11.8%.

Yuasa claims in [20] that the time for saving the root set of a thread can be made sufficiently short by using block transfer mechanisms. Our experience however shows that the atomic copying of the stack takes too long to keep the jitter reasonably low for tasks with sub-millisecond periods.

A later approach by Yuasa [21] introduces a *return barrier*. In a first step, the garbage collector scans the topmost stack frames of all threads atomically. Then it continues to scan one frame at a time. When a thread returns to a frame that has not yet been scanned, it scans it by itself. Return instructions consequently carry an overhead for the respective check. Furthermore, the proposed policy makes it difficult to compute tight WCET bounds, because it is difficult to predict when a scan by a thread is necessary. A critical issue is also that the topmost frames of *all* threads have to be scanned in a single atomic step. The worst case blocking time therefore increases with the number of threads. An overhead of 2 to 10% percent due to the necessary checks for the return barrier is reported in [21]. Depending on the configuration, 10 to 50 μs were measured as worst case blocking time on a 200 MHz Pentium Pro processor.

Cheng et al. propose a strategy for lowering the overhead and blocking of stack scanning in [5]. The mutator thread marks the activated stack frames and the garbage collector scans only those frames which have been activated since the last scan. However, this technique is only useful for the average case. In the worst case, it is still necessary to scan the whole stack atomically.

3. PREEMPTIBLE ROOT SCANNING

Due to the volatile nature of a thread's stack, the garbage collector and the mutator thread must cooperate for proper scanning. If a thread executes arbitrary code while its stack is scanned, the consistency of the retrieved data cannot be guaranteed. Therefore, a thread is usually suspended during a stack scan. In order to ensure the consistency of the root set, the stack is scanned atomically to avoid preemption of the garbage collector and inhibit the execution of the respective thread.

When the garbage collection thread scans a stack it is not allowed to be preempted by that thread. However, as the stack is thread local, preemption by any other mutator thread is not an issue. Therefore, a thread will only suffer blocking time due to the scanning of its own stack. A high priority thread, which has probably a shallow call tree, will not suffer from the scanning of deeper stacks of more complex tasks. The protection of the scan phase can be achieved by integrating parts of the garbage collection logic with the scheduler. During stack scanning, only the corresponding mutator thread is blocked.

We generalize this idea by moving the stack scanning task to the mutator threads. Each thread scans its own stack at the end of its period. In that case mutual exclusion is trivially enforced: the thread performs either mutator work or stack scanning. The garbage collector initializes a garbage collection period as usual. It then sets a flag to signal the threads that they shall scan their stacks at the end of their period. When all threads have acknowledged the stack scan, the garbage collector can scan the static variables and proceed with tracing the object graph. Why the static variables are scanned after the local variables is discussed in Section 3.1.4.

By using such a scheme, it is not necessary to enforce the atomicity of a stack scan. Furthermore, the overhead for a stack scan is low; at the end of each period, the stack is typically small if not even empty. Such a scheme also simplifies exact stack scanning, because stack scanning takes place only at a few predefined instants. Instead of determining the stack layout for every point at which a thread might be preempted, it is only necessary to compute the layout for the end of a period. The required amount of data is reduced considerably as well, which lowers the memory overhead for exact stack scanning.

3.1 Correctness

We identified three properties which must be fulfilled to ensure the correctness of our proposed garbage collection scheme:

- The local root set remains unmodified during scanning (Property 1).
- Modification of the object graph during scanning does not void correctness (Property 2).
- Migration of references between local root sets does not void correctness (Property 3).

Property 1 refers to the situation, where a thread moves a reference from a not-yet-scanned local variable to a scanned local variable and clears the not-yet-scanned variable afterwards. In such a case, the respective root reference would be lost.

Property 2 addresses the fact that two local root sets are scanned at different times. They do not necessarily correspond to the same state of the object graph. It must be shown that this does not invalidate the correct view of the object graph at mark time. This allows threads to execute between the scanning of two local root sets.

Symbol	Description
G	Object graph
τ_i	Thread i
$V(G)$	Nodes of an object graph G (objects)
$E(G)$	Edges of an object graph G (references)
$R(G)$	Root set of an object graph G, $R(G) \subseteq V(G)$
$\rho(G)$	Reachability function, set of reachable nodes in G
$\mu(G)$	Marking function, approximation of $\rho(G)$

Table 1: Symbol descriptions

Property 3 is similar to Property 1, but now a reference is moved between local root sets. This is also an issue if the stack of each thread is scanned atomically. The error scenario for this property is when a reference is transferred from a not-yet-scanned stack to a scanned stack and removed from the not-yet-scanned stack afterwards.

In the following, we will sketch the proofs why the Properties 1-3 are fulfilled in our proposed garbage collection scheme. The symbols we use throughout this section are described in Table 1. An object graph can also be written as tuple of its nodes, edges and root set: $G = \langle V(G),\ E(G),\ R(G)\rangle$. We consider only the objects that are referenced by local variables as part of the root set $R(G)$. Objects which are referenced by static variables are handled separately, as described in Section 3.1.4.

We define the operations $\cup$ and $\supseteq$ on object graphs as component-wise application of these operations to the nodes, edges and root sets of two object graphs:

$$G \cup G' := \langle V(G) \cup V(G'),\ E(G) \cup E(G'),\ R(G) \cup R(G')\rangle$$
$$G \supseteq G' := V(G) \supseteq V(G') \wedge E(G) \supseteq E(G') \wedge R(G) \supseteq R(G')$$

We define a relation $G \bowtie G'$ ("G bow tie G'") of two graphs as follows:

$$G \bowtie G' \Leftrightarrow \forall v_1 \in V(G) \cap V(G') : \langle v_1, v_2\rangle \in E(G) \Rightarrow \langle v_1, v_2\rangle \in E(G'),\ v_2 \in V(G') \wedge \langle v_1, v_2\rangle \in E(G') \Rightarrow \langle v_1, v_2\rangle \in E(G),\ v_2 \in V(G)$$

We call two graphs G and G' *out-edge consistent* if $G \bowtie G'$. This relation holds if out-edges of shared nodes are present in both graphs. The out-edge consistency of object graphs is important when reasoning about how different threads see a shared object graph. Assuming the out-edge consistency of sub-graphs greatly simplifies formal handling.

Due to the relatively loose requirements on data consistency of the Java memory model [9], two threads do not necessarily share the same view of an object.[2] While one thread assumes that a field obj.f references an object x, another thread may assume that the same field references an object y. Such a situation is problematic for a garbage collector, because it would leave either object x or object y unvisited.

Such inconsistencies originate from the fact that threads are allowed to cache data locally. On uniprocessors, cache coherence is not an issue – all threads share the same cache – but thread-local registers may be used to store reference fields. As these registers are scanned for the computation of the local root set of a thread, it is ensured that references cached in registers are visited as well as references stored in the heap. It is therefore safe to assume that all threads have an out-edge consistent view of the object graph.

On multi-processors, cache coherence must be ensured to allow consistent tracing of the object graph. It is then also safe to assume the out-edge consistency of the threads' views of the object graph. It is beyond the scope of this paper how this cache coherence can be achieved efficiently.

The reachability function is formally defined as follows:

$$\rho(G) := \{v \in V(G) \mid \exists \text{ a chain of edges } e_1, \ldots, e_n \in E(G) : e_1 = \langle r, v_1\rangle, e_2 = \langle v_1, v_2\rangle, \ldots, e_n = \langle v_{n-1}, v\rangle,\ r \in R(G)\}$$

In other words, a node v is reachable, if there exists a path from a root node $r \in R(G)$ to v.

The reachability function ρ has three important properties:

- ρ is *monotone*:
  $G \supseteq G' \Rightarrow \rho(G) \supseteq \rho(G')$
- ρ is *closed*:
  $G' = \langle V(G),\ E(G),\ R(G) \cup \{v\}\rangle,\ v \in \rho(G) \Rightarrow \rho(G) = \rho(G')$
- ρ is *distributive* for out-edge consistent graphs:
  $G \bowtie G' \Rightarrow \rho(G \cup G') = \rho(G) \cup \rho(G')$

The monotony of ρ is a fairly intuitive property: adding a node or an edge to a graph cannot make fewer nodes reachable. ρ is closed in the sense that adding a reachable node to the root set does not change the set of reachable nodes. This property becomes clearer when considering that root nodes are just known to be reachable a priori, but are not special in any other way.

The distributivity of ρ allows to determine the set of reachable node globally or locally. Consider two threads τ and τ' and their views of the object graph, G and G'. It is possible to compute $R(G) \cup R(G')$ and start tracing from that root set; it is also possible to compute $\rho(G)$ and $\rho(G')$ independently and merge the results. Distributivity ensures that the final result is the same for both approaches.

An important observation is that a thread may only access objects that are reachable, or create new objects. This insight may seem trivial; still, it should be kept in mind for the following considerations.

There are four fundamental operations on an object graph, which are shown in Table 2. We use the following formal definitions for these graph operations:

$$f_{new}(G) := \langle V(G) \cup \{v\},\ E(G),\ R(G)\rangle,\ v \notin V(G)$$
$$f_{load}(G) := \langle V(G),\ E(G),\ R(G) \cup \{v\}\rangle,\ v \in V(G)$$
$$f_{kill}(G) := \langle V(G),\ E(G),\ R(G) \setminus \{v\}\rangle$$
$$f_{write}(G) := \langle V(G),\ (E(G) \setminus \{\langle v_1, v_2\rangle\}) \cup \{\langle v_1, v_3\rangle\},\ R(G)\rangle$$

It must be noted that each thread can only operate on its local view of the object graph. Therefore, f_{load} and f_{kill} can only modify the local root set of a mutator thread. Furthermore, these are "minimal" definitions; they have to be extended for actual garbage collection algorithms, e. g., to model a write barrier.

Operation	Description
f_{new}	Create a new object
f_{load}	Add a reference to the root set
f_{kill}	Remove a reference from the root set
f_{write}	Replace an edge with a new edge

Table 2: Description of operations

[2] Proper synchronization of course eliminates coherence problems, but the authors consider correctness of synchronization to be an unreasonably strong precondition. Flawed synchronization should not cause a failure of the garbage collector.

We assume that any changes to the shape of the object graph retain the out-edge consistency of the local views. As already mentioned, this is a safe assumption for uniprocessors. For multiprocessors, this assumption is only safe if cache coherence is ensured.

3.1.1 Property 1: Atomicity

The runtime stacks of any two threads are disjoint – otherwise they could not execute independently of one another. There is no way how a Java thread can access the stack of any other thread. Only JVM-internal threads like the garbage collection thread can access the stack of another thread. Therefore, the execution of a Java thread cannot modify the stack of a different thread.

If the garbage collector scans a thread's stack, it is hence not necessary to enforce absolute atomicity. It is only necessary to prevent the thread whose stack is scanned from executing – all other threads cannot modify the stack of this thread. Such a strategy needs support from the scheduler and does not eliminate blocking completely. However, blocking can be reduced considerably, because threads only need to wait while their own stack is scanned and high frequency threads typically have a shallow stack.

It also follows that a thread may be preempted while scanning its own stack without compromising the consistency of the scanned data. When the thread continues execution, the stack is unchanged and the local root set is the same as if the thread had not been preempted. It is not necessary to enforce atomicity if a thread scans its own stack.

3.1.2 Property 2: Independence

Due to the distributivity of ρ, it is sufficient to scan the stack of one thread at a time to determine the global set of reachable nodes. However, it needs to be shown that the actions of one thread do not interfere with the view of the object graph of another thread.

f_{new} allocates a new object, which is not yet visible to other threads; it can therefore be considered as local action. f_{load} and f_{kill} only modify the local root set and hence are local actions as well. Consequently, threads can only communicate through the f_{write} operation. This operation however retains the out-edge consistency, i.e., if one thread changes the shape of the object graph, other threads see the updated object graph. Therefore, it is not necessary for one thread to know anything about the root set or the actions of a different thread for the proper computation of ρ. The root sets can therefore be scanned independently of one another.

3.1.3 Property 3: Consistency

It is necessary that the local root sets and the object graph are consistent such that no less than the actually reachable nodes are marked during tracing. It must be ensured that the fundamental graph operations cannot break this consistency. Please note that the following reasoning refers to the root scanning phase and not to the marking phase. This means that no node is *black* in terms of Dijkstra's tricolor abstraction [6].

In order to keep the root set consistent for f_{new}, it is necessary to add references to new objects to the root set. Otherwise, recently created objects could erroneously appear unreachable. In terms of the tricolor abstraction, this means that new objects are allocated gray during root scanning. The f_{new} operation therefore has to be extended as follows:

$f_{new}(G, type) := \{$
  $v :=$ GC.allocate($type$);
  GC.markGray(v);
  $V(G) := V(G) \cup \{v\}$;
$\}$

f_{load} can only add references to the root set which are already reachable. Due to the closed nature of ρ, this does not change the set of reachable nodes. If a snapshot of the object graph from the beginning of the garbage collection cycle is maintained, these operations have no effect.

f_{kill} removes a reference from the root set and can therefore change the set of reachable nodes. If an object is still reachable after this operation, it must be reachable through a path which starts at a different root node. As no node is black during root scanning, this path will be encountered by the marking function. If the most recent state of the object graph is traced, it is not necessary to take measures to prevent the loss of a root reference.

If it can be shown that the side effects of f_{write} allow that both the snapshot of the object graph and the current object graph are traced, both f_{load} and f_{kill} do not require any special action.

We define a *history graph* H, which subsumes changes to an object graph. With G being an object graph and $f_0 \dots f_n \in \{f_{new}, f_{load}, f_{kill}, f_{write}\}$ being a sequence of graph operations, it is defined as

$$V(H) := V(G \cup f_0(G) \cup f_1(f_0(G)) \cup \dots \cup f_n(\dots(f_0(G))))$$
$$E(H) := E(G \cup f_0(G) \cup f_1(f_0(G)) \cup \dots \cup f_n(\dots(f_0(G))))$$
$$R(H) := R(f_n(\dots(f_0(G))))$$

Such a history graph safely approximates the shape of both the initial object graph and the current object graph. Therefore, the considerations for object graph operations above apply to a history graph. An implementation of f_{write} which allows to safely approximate a history graph is consequently sufficient to maintain the correctness of ρ.

In [1], a *double barrier*, which saves both references on an assignment, is suggested to maintain the consistency of the root set. The reasoning behind this is that references could otherwise migrate from the local root set of one thread to the root set of a different thread without being noticed. The critical operation for this assumption is when a reference migrates from a not-yet-scanned local root set to a local root set which already has been scanned. However, a thread may only access reachable objects and can therefore only add references to its root set which are already reachable. Therefore, the respective reference is also already reachable from the scanned root set. In a history graph, this reference remains visible, even if the addition of the reference to the root set is ignored.

We now show that a snapshot-at-beginning barrier as suggested in [20] is sufficient to approximate a history graph. Together with the independence of local root scans and the considerations on the graph operations above, this proves that such a double barrier is indeed not necessary.

We use the following abstract definition of f_{write}, which models the Yuasa snapshot-at-beginning write barrier:

$f_{write}(G, \langle v_1, v_2\rangle, \langle v_1, v_3\rangle) := \{$
  if (color(v_2) = white) {
    GC.markGray(v_2);
  }
  $E(G) := (E(G) \setminus \{\langle v_1, v_2\rangle\}) \cup \{\langle v_1, v_3\rangle\}$;
$\}$

This definition may look unusual when comparing it to actual implementations. It is however just a translation of such an implementation to the terminology used throughout this paper and does not add any special semantics. The barrier shades references that are overwritten gray. The overwritten reference is therefore visible for the tracing algorithm; in effect, a virtual snapshot of the object graph is retained.

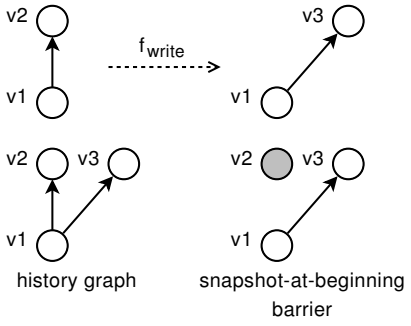


Figure 1: History graph and snapshot-at-beginning barrier

The marking function μ is an algorithm to compute the set of reachable nodes. Starting from the root set, the object graph is traversed, until no new objects can be reached. It can be assumed that the marking function μ safely approximates the reachability function ρ if no graph transformations occur. This assumption is fundamental for *any* tracing garbage collection algorithm and must be proven for the correctness of any garbage collector, independent of root scanning and even for stop-the-world garbage collectors. Note that our considerations apply to the root scanning phase and not marking itself. Our goal is to show that the root set is computed correctly. Proving the correctness of concurrent marking is a different issue; such proofs can be found in [6] or [20].

f_{load} and f_{kill} change only the root set of a graph; as the root set of the history graph is the root set after any transformation, μ trivially approximates the history graph w. r. t. these operations. Shaded allocation of new objects ensures that f_{new} is handled correctly. This leaves to be shown that the write barrier indeed allows the approximation of a history graph.

Figure 1 shows the effect of an $f_{write}(G) = G'$ operation. The history graph includes all encountered edges, i.e., the replaced edge $\langle v_1, v_2 \rangle$ and the new edge $\langle v_1, v_3 \rangle$. Consequently, the following equation holds for $H = G \cup G'$:

$$v_1 \in \rho(G) \Rightarrow (v_2 \in \rho(H) \wedge v_3 \in \rho(H)) \tag{1}$$

The snapshot-at-beginning barrier does not retain the edge $\langle v_1, v_2 \rangle$, but it marks v_2 to be traced. The appropriate equation is:

$$(v_1 \in \mu(G) \Rightarrow v_3 \in \mu(G')) \wedge v_2 \in \mu(G') \tag{2}$$

which can be reformulated as

$$(v_1 \in \mu(G) \Rightarrow v_2 \in \mu(G') \wedge v_3 \in \mu(G')) \wedge v_3 \in \mu(G') \tag{3}$$

These equations entail that μ induces a superset of ρ if operating on the same graph. Therefore, a snapshot-at-beginning barrier is sufficient to ensure the correctness of incremental garbage collection.

3.1.4 Static Variables

The considerations above cover only the scanning of thread-local roots, but left out static variables. They can be modeled with an immutable root, which points to a virtual array that contains the static variables. This virtual array can then be handled like any other object and the scanning of static variables becomes part of the marking phase. As marking has to take place after root scanning, static variables have to be scanned after the local root sets.

This is not an arbitrary limitation – it is easy to construct an example where scanning static references before scanning local variables breaks the consistency of a garbage collector that uses a snapshot-at-beginning write barrier. Consider the case where during the scanning of static variables a reference is transferred from a local variable to a static variable which has already been scanned. The value of the local variable might be lost until it is scanned, and the new value of the static variable is not visible to the garbage collector. The variable already has been scanned, and the snapshot-at-beginning barrier retains the old value, but does not treat the new one. Therefore, the respective object may erroneously appear unreachable to the garbage collector.

In contrast, we were not able to construct an example where a reference is lost if the scanning of static variables takes place *after* the scanning of local variables. As it can then be regarded as part of the marking phase, the correctness proof of the snapshot-at-beginning algorithm [20] apply to this part of the algorithm – it is simply impossible to find a counterexample. Actually, the theoretical findings presented above were sparked by our unavailing search for such an example.

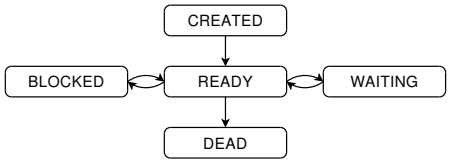


Figure 2: Thread Model

3.2 Execution Time Bounds

Now that we have established the correctness of our approach, we can analyze the effects on the timing behavior of the garbage collection thread. We found two solutions to apply the theoretical results: The first solution can be applied only to periodic tasks, while the second solution can be applied to sporadic tasks as well.

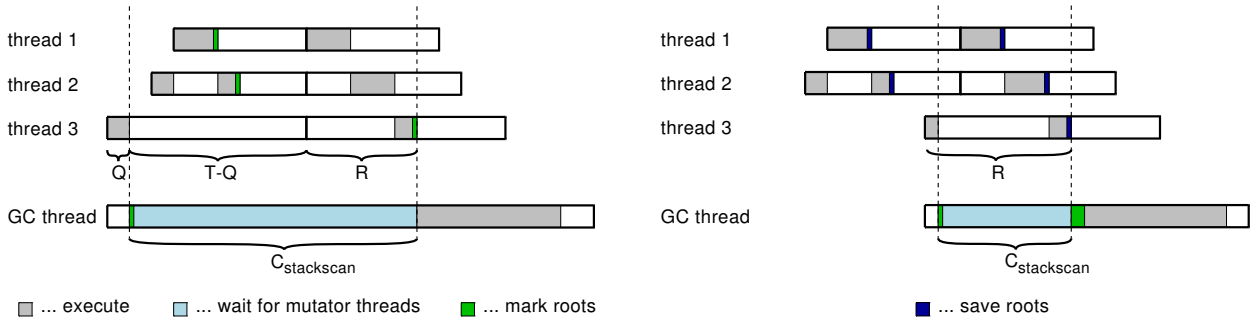
3.2.1 Thread Model

We assume that all threads are either periodic or have at least a known deadline. This is a reasonable assumption for real-time threads: it is impossible to decide whether a task delivers its results on time if no deadline or period is known.

The thread model has five states: CREATED, READY, WAITING, BLOCKED and DEAD. Initially, a thread is in state CREATED. When a thread gets available for execution, it goes to the READY state. When it has finished execution for a period it becomes WAITING. At the start of the next period, it goes to the READY state again. If a thread terminates, it becomes DEAD. Threads are in state BLOCKED while they wait for locks or I/O operations. The time between the instant at which a thread becomes READY until it goes to state WAITING must be bounded – if it is not WAITING when its deadline arrives, it has missed the deadline. Figure 2 visualizes the possible state transitions of the thread model.

For the calculation of the execution time bounds, we assume that threads scan their stack when they become WAITING. For periodic tasks in the Real Time Specification for Java (RTSJ) [4], this can be done implicitly when waitForNextPeriod() is invoked. There is no need to change the application code. If no such method needs to be called by tasks, the scanning can be integrated into the scheduler. In the current version of the RTSJ, sporadic threads do not invoke such a method; their stack is however empty when they do not execute, which in turn makes root scanning trivial. The overhead for stack scanning of course has to be taken into account for calculating the WCET of tasks.

We assume that the garbage collection thread runs at the lowest priority in the system. On the one hand, a garbage collector usually



(a) Solution for Periodic Tasks

(b) Generalized Solution

Figure 3: Visualization of the WCET for root scanning

has a long period (and deadline), compared to other real-time tasks. It follows from scheduling theory that it should have a low priority [11]. On the other hand, a real-time scheduler does not even need to be aware of the garbage collector to achieve full mutator utilization in such a setting.

3.2.2 Solution for Periodic Tasks

For periodic threads, the time between two releases is known and the time between two successive calls of waitForNextPeriod() is bounded. For this solution, the individual tasks push the references of the local root set onto the mark stack of the garbage collector if an appropriate flag is set. The garbage collector must wait until all tasks have acknowledged the scan before it can proceed. In the worst case, a task has become WAITING very early in its period when the garbage collector starts execution and becomes WAITING very late in its next period.

Let R_i be the worst case response time of a thread τ_i, Q_i its best case response time and T_i its period. The response time of a task is the time between the instant at which a thread becomes READY until it goes to the WAITING state again. $C_{stackscan}$ is the worst case time until all threads have scanned their local root set and the garbage collector may proceed. $C_{stackscan}$ can be computed as follows:

$$C_{stackscan} = \max_{i \geq 0}(T_i - Q_i + R_i) \tag{4}$$

Figure 3(a) visualizes the formula above. It shows that the worst case for two responses of a thread is $T - Q + R$. Consequently, this is the longest time the garbage collector must wait for this thread.

To avoid the computation of the best and worst case response times – especially the former is typically unknown –, this can be simplified to

$$C_{stackscan} = 2T_{max} \tag{5}$$

$C_{stackscan}$ has to be added to the response time of the garbage collection thread; the impact of this delay depends on the thread periods. If the minimal period of the garbage collector is far greater than the periods of the mutator threads, the relative impact is small. If there is some slack between the minimal and the actual garbage collection period, the effect can probably be hidden. If the maximum period of the mutator threads is relatively long, this may have a notable effect on the minimal garbage collection period.

3.2.3 Generalized Solution

The considerations for periodic tasks cannot be applied to sporadic tasks in the general case. For sporadic tasks, the minimum inter-arrival time is known, but usually not the maximum inter-arrival time. Therefore, the worst case time until the garbage collector may proceed is potentially unbounded.

The stack of a thread is only modified if the respective thread executes. Therefore, the garbage collector can reuse data from previous scans and only needs to wait for threads which may have executed since their last scan. These are – apart from the initialization and destruction of threads – the threads which are not in state WAITING.

We adapt the root scanning scheme such that threads save their stack on every call of waitForNextPeriod() to a *root array*. For WAITING threads, the content of the root array from their last scan is used by the garbage collector; for all other threads, the garbage collector waits until they have updated their root array. With this scheme, it is sufficient to take into account the worst case response time for the execution time of stack scanning.

$$C_{stackscan} = R_{max} \tag{6}$$

For schedulers which never execute lower priority threads if a higher priority thread is READY and which do not support blocking operations, the garbage collector will never encounter any threads which are not WAITING. Consequently, it is never necessary to wait for any thread to scan its stack and

$$C_{stackscan} = 0 \tag{7}$$

The downside of this approach is that a dedicated memory area is needed to save the roots of the individual threads. It is not possible anymore to let the mutator thread push its root set onto the mark stack. To avoid blocking in this scheme, it is necessary to use two memory areas for each thread to allow for double buffering. If the maximum number of roots is unknown, each of these areas occupies as much memory as the stack.

For this strategy, the overhead for completing the root scanning is larger on the garbage collectors side. This is due to the fact that the garbage collector itself has to push the references onto the mark stack. On the threads' side, the overhead is slightly smaller, because the content of the stack only has to be transferred to the root array, without performing any computations.

An advantage of this strategy is that $C_{stackscan}$ is considerably lower – the increased overhead for scanning is most likely far smaller than the time that is spent on waiting for the other threads. It is mandatory to use such a scheme for sporadic tasks; applying it to periodic tasks as well allows to trade off time to wait for a root scan with additional memory consumption.

Figure 3 compares the worst case scenarios of the solution for periodic tasks and the generalized solution. For the generalized solution, the threads save their stack in a root array at the end of

each period. The garbage collector only has to wait for threads which have executed since their last scan. In Figure 3(b), thread 3 is BLOCKED when the garbage collector starts execution. Therefore, the garbage collector only has to wait for this thread. In the worst case, this waiting time equals the maximum response time.

3.2.4 Discussion

As pointed out in [12] and [15], the allowable allocation rates and the minimum garbage collection period are related. Increasing the period of the garbage collector effectively lowers the allowable allocation rates. The proposed solutions introduce a waiting time for the garbage collector and therefore may make it necessary to increase its period.

However, it is possible to mix root scanning strategies to find an optimal solution. For high frequency threads, jitter is usually very important, and the waiting time of the solution for periodic threads may be negligible. For medium frequency threads, the generalized solution with an impact in the order of one period may be a better trade-off. Low frequency threads are probably less sensitive to jitter and root scanning by the garbage collector may not hurt them. It is however not possible to propose a generic solution to this problem without knowledge about the properties of the whole system.

4. EVALUATION

We used the Java Optimized Processor (JOP) [14] for the evaluation. The processor is implemented in an FPGA and runs at 100 MHz. The platform we used features 1 MB of SRAM with 15 ns access time. The garbage collection algorithm used by JOP [17] is an incremental garbage collector, with a snapshot-at-beginning write barrier [20]. It is based on the copying collector by Baker [3], but uses a forwarding pointer placed in an *object handle* to avoid the costly read barrier. While the absolute times reported in this section are of course platform dependent, the evaluated concepts are not specific to JOP and its garbage collector.

A special feature of our garbage collector is hardware support for interruptible copying [16]. A special hardware module translates accesses to objects and arrays that are copied, such that copying is transparent to the mutator threads. Accesses to fields that have not been copied yet are directed to the source location; accesses to already copied fields are directed to the destination location. Only single words need to be copied atomically. This allows even large arrays to be copied without introducing blocking times.

For jitter measurements, we used 6 different tasks. Deadline monotonic priority ordering is used to determine the tasks' priorities; unless otherwise stated, the deadline of a task equals its period. The task properties are described in the following and subsumed in Table 3. The figures presented in Table 4 were obtained by measuring the maximum release jitter of the highest priority thread during a run of more than one hour. For the measurements, we slightly modified the periods of the threads. We used prime numbers (e.g., 1009 μs instead of 1000 μs) to avoid a regular phasing of the threads, which could have led to too optimistic results.

The most important thread w. r. t. the measurements is the high-frequency task τ_{hf} with a period of 100 μs. It computes its own release jitter and does nothing else. This task has the highest priority of all tasks and all jitter figures in this section refer to the release jitter of this thread.

Two more threads, τ_p and τ_c, implement a producer/consumer pattern. τ_p produces one object every millisecond and τ_c consumes the available objects every 10 milliseconds. A simple list is used to pass the objects from τ_p to τ_c. These threads have the second- and third-highest priorities in the system. $\tau_{p'}$ is a slightly modified version of τ_p, which does not actually allocate an object. Instead, it emulates the blocking behavior of new, i.e., it contains a synchronized block which requires as long as the largest synchronized section in the implementation of new. This task is used to distinguish between the impact of synchronization of threads and the impact of garbage collection.

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	6
τ_p	1 ms	1 ms	5
τ_c	10 ms	10 ms	4
τ_s	15 ms	15 ms	3
τ_{log}	1000 ms	100 ms	2
τ_{gc}	200 ms	200 ms	1

Table 3: Thread properties of the test program

τ_s is a thread which occupies the stack such that it is not empty when waitForNextPeriod() is invoked. Consequently, a strategy as proposed by Schoeberl and Vitek in [17] cannot be used if this thread is part of the task set. The period of τ_s is 15 ms.

To record the measurements, we used a logging thread τ_{log} with a period of 1000 ms and a deadline of 100 ms. The garbage collection thread, τ_{gc}, has a period of 200 ms and is consequently the lowest-priority thread in the system.

We used various combinations of the tasks described above to evaluate the different root scanning strategies. The simplest task set comprises only the high-frequency task τ_{hf}. The jitter for this task indicates if the system is able to run this task at the requested frequency at all. The task sets $\{\tau_{hf}, \tau_{log}\}$ and $\{\tau_{hf}, \tau_s, \tau_{log}\}$ are used to measure the jitter due to task switches. As no garbage collection takes place and none of the tasks uses synchronized blocks, this is the only source of jitter for these two task sets. The jitter due to the combination of task switches and synchronized blocks are evaluated through the task sets $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_{log}\}$ and $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_s, \tau_{log}\}$. The other task sets present in Table 4 are used to evaluate the impact of garbage collection on τ_{hf}. τ_s has a relatively deep stack – task sets which contain τ_s therefore challenge the root scanning phase. τ_p and τ_c produce and consume objects and test the impact of the actual garbage collection on τ_{hf}.

The first row in Table 4 shows the trivial task set $\{\tau_{hf}\}$. The results for the task sets without garbage collection can be found in the next four rows. The bottom four rows present the results for the task sets with garbage collection.

We evaluated five different root scanning strategies. The strategy labeled *base* in Table 4 scans the stacks of all threads in one atomic step. The *single* strategy scans one stack at a time atomically. The strategy as proposed by Schoeberl and Vitek [17], which assumes that the thread stacks are empty at the time of root scanning, is labeled *empty*. The *scan* and *save* strategies implement the solution for periodic tasks and the generalized solution as described in Section 3.2. For the *scan* strategy, tasks push their local root set onto the mark stack at the end of their period. For the *save* strategy, tasks save their stack into root arrays, and the garbage collector pushes the references onto the mark stack.

4.1 Results

The results in Table 4 show that the *base*, *single* and *empty* strategies behave identical if no garbage collection takes place. This is no surprise, because the implementation of the scheduler is the same. The *scan* strategy behaves slightly worse, and the *save* strategy adds up to 27 μs of jitter (73 μs for *base*, *single* and *empty* compared to 100 μs for *save* in row 5 of Table 4). As we made only minimal changes to the scheduler, we had expected only a

Thread Set						Jitter (μs)				
τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	τ_{gc}	base	single	empty	scan	save
✓						0	0	0	0	0
✓				✓		41	41	41	55	47
✓			✓	✓		68	68	68	66	57
✓	✓[a]	✓		✓		67	67	67	67	70
✓	✓[a]	✓	✓	✓		73	73	73	80	100
✓				✓	✓	321	110	57	56	56
✓			✓	✓	✓	488	180	–	70	65
✓	✓	✓		✓	✓	514	120	71	77	93
✓	✓	✓	✓	✓	✓	685	182	–	91	106

Table 4: Jitter Measurements

[a] $\tau_{p'}$, a slightly modified version of τ_p, was used for this measurement.

smaller deviation in the results. We observed that small changes in the application code sometimes have a considerable effect on the performance of the method cache [13] of JOP. However, further research will be necessary to find out if this is the actual source of the jitter increase. In the following, we do not attribute this jitter to garbage collection itself.

If garbage collection takes place, the *base* strategy performs worst w. r. t. the release jitter of τ_{hf}. The jitter introduced by this strategy is several times larger than for the other strategies. 685 μs were measured as worst case jitter. The high jitter is caused by the atomic scan of all stacks; it increases with the number of threads.

The *single* strategy yields less jitter than *base*, but still more than the other strategies. The jitter for this strategy depends on the size of the largest thread stack; it is highest if τ_s is part of the thread set. Up to 182 μs jitter were observed during our experiments.

The *empty*, *scan* and *save* strategies induce a similar amount of jitter. However, *empty* cannot be applied if τ_s is part of the task set, because it cannot handle stacks which are not empty at the time of root scanning. A side effect of *scan* is that it implicitly extends the garbage collection period to 1000 ms. The reason for this is that the garbage collector cannot proceed until all other threads have run. τ_{gc} always has to wait until τ_{log} has executed once and therefore is locked to the frequency of the latter. The *save* strategy solves the problems of *empty* and *scan* at the expense of an increased memory consumption.

4.2 Discussion

Our goal was to minimize the impact of garbage collection on other threads. The figures in Table 4 show that a considerable amount of jitter is caused by scheduling and synchronized sections. Scheduling introduces jitter of 41 to 68 μs; scheduling and synchronized blocks together result in a jitter of 67 to 73 μs. For the *base* and *single* strategy, 50 to 600 μs are added to the jitter by the garbage collection thread. The *empty*, *scan* and *save* strategies perform considerably better in this regard.

The *scan* strategy adds 1 to 11 μs, when comparing the task sets with and without garbage collection. The *save* strategy adds 23 μs of jitter to the task set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}\}$ and less than 10 μs to the other task sets. As blocking was eliminated from the root scanning phase, this can obviously not be the reason for the increased jitter. We assume three possible sources for the jitter increase: increased scheduling overhead, degraded performance of the instruction cache and imprecise measurements. On the one hand, longer measurements could have increased the measured jitter for some test cases. On the other hand, it is possible that the worst case behavior of some parts of the code only occurs if garbage collection actually takes place. Still, the jitter introduced by the two new root scanning strategies is considerably lower than the jitter introduced by scheduling and synchronization. Future work will have to show how far these two sources of jitter can be eliminated.

5. CONCLUSION AND OUTLOOK

We investigated the root scanning phase of garbage collection on a theoretical basis and could prove three important properties: First, that atomicity for stack scanning is only necessary w. r. t. the thread whose stack is scanned. Second, that atomicity is not required at all if mutator threads scan their own stack. And third, that a snapshot-at-beginning write barrier is sufficient to allow complete decoupling of local stack scans.

Furthermore, we provided two approaches how these theoretical properties can be utilized and showed the implications on the execution time of a garbage collector. The first approach can be applied only to periodic tasks and delays garbage collection by up to two times the largest task period. The second approach is more general and has a smaller impact on the execution time of the garbage collector, but has a higher memory overhead.

An evaluation of the two new approaches to root scanning confirmed the theoretical results. Jitter of high priority threads, which can be attributed to garbage collection, could be reduced considerably. The impact of the new root scanning strategies on the jitter due to scheduling and synchronization however still needs to be analyzed.

Future work will investigate if a tighter coupling of scheduling and root scanning is profitable. Merging the root arrays of the generalized solution with the memory areas for the thread contexts could lower the memory consumption without impairing the performance.

Exact stack scanning has not been handled in this paper. The proposed solutions lower the overhead for exact scanning, but tools to make use of this need to be developed.

Future work will also have to extend the proposed solutions to multi-processor systems. We are confident that the theoretical basis is applicable to such systems as well, but actual implementations may offer new obstacles as well as new opportunities.

6. ACKNOWLEDGEMENT

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

7. REFERENCES

[1] J. Auerbach, D. F. Bacon, B. Blainey, P. Cheng, M. Dawson, M. Fulton, D. Grove, D. Hart, and M. 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.

[2] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, Jan. 2003. ACM Press.

[3] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[6] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, Nov. 1978.

[7] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, Jan. 1994. ACM Press.

[8] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, Jan. 1993.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.

[10] R. Henriksson. Scheduling real-time garbage collection. In *Proceedings of NWPER'94*, Lund, Sweden, 1994.

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

[12] S. G. Robertz and R. Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[13] M. Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[14] M. Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[15] M. 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, Apr. 2006.

[16] M. Schoeberl and W. 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)*, 2008.

[17] M. Schoeberl and J. Vitek. Garbage collection for safety critical Java. In *Fifth International Workshop on Java Technologies for Real-Time Systems (JTRES)*, pages 85–93, Vienna, Austria, Sept. 2007. ACM Press.

[18] F. Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, Apr. 2001.

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

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

[21] T. Yuasa. Return barrier. In *Proceedings of the International Lisp Conference 2002*, 2002.