Concurrent, Parallel, Real-Time Garbage-Collection

Dr. Fridtjof Siebert

aicas GmbH
Haid-und-Neu-Str. 18
76131 Karlsruhe, Germany
siebert@aicas.com

Abstract

With the current developments in CPU implementations, it becomes obvious that ever more parallel multicore systems will be used even in embedded controllers that require real-time guarantees. When garbage collection is used in these systems, parallel and concurrent garbage collection brings important performance advantages in the average case. In a real-time system, however, guarantees on the GC's performance in the worst case are required.

This paper explains how the single-CPU real-time GC of the Java implementation JamaicaVM was changed to make it a hard real-time garbage collector that is parallel and concurrent. Parallel means that an arbitrary number of CPUs may perform GC work in parallel, while concurrent means that the GC work can be performed concurrently to the application code without pre-empting the application. In addition, the single units of work that this garbage collector has to perform are very small and uniform and the total amount of GC work is bounded by a function of the heap size, such that it becomes possible for any application that has a bounded amount of reachable memory to run the GC work such that sufficient GC progress can be ensured for the application never to run out of heap space.

Categories and Subject Descriptors C.3 [*Computer Systems Organization*]: SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS—Real-time and embedded systems; D.4.7 [*Software*]: OPERATING SYSTEMS—Organization and Design: Real-time systems and embedded systems

General Terms algorithms, languages, performance, reliability

Keywords multicore, parallel, concurrent, real-time, garbage collection, Java

1. Introduction

The emergence of multicore computer architectures will have a profound effect on the software development process and the implementation of programming languages. With multicore systems, parallel systems become the norm even for low-end computers such as embedded controllers. Languages that perform automatic memory management require garbage collector implementations that make use of this parallel processing power.

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

The goal of such parallel garbage collector implementations is typically an increased average performance of the memory management system, which is required to keep up with the increase in mutator performance on parallel systems. Systems with real-time requirements, such as embedded controllers, however, need proven upper bounds on the worst case behaviour of the implementation. For a parallel garbage collector, this means that a bound on the worst-case performance has to be found.

1.1 Terminology

In the memory management community, the term *real-time* has often been used in a way that is much more relaxed than its use in the real-time community. In the real-time community, a real-time system is a system that has hard deadlines that must not be missed. It is in this strict sense that real-time is understood within this paper: proven upper bounds worst-case performance ar needed.

The literature categorises garbage collectors (GCs) as incremental, concurrent or parallel. These categories are not disjoint, e.g., a parallel garbage collector may also be concurrent.

An *incremental* GC is a GC that can operate incrementally, i.e., it can perform a full garbage collection cycle in parts and allow the application to perform work in between these incremental garbage collection steps. A *concurrent* GC can operate in parallel to the application. Finally, a *parallel* GC is a GC that can use several processors simultaneously to perform its work in parallel. A parallel GC may be non-concurrent. Also, a concurrent GC can be non-parallel.

For a GC to be real-time, it is required that the performance of the application threads is predictable, in particular, there must not be any unbounded pre-emption due to GC work and the GC must ensure that it reclaims memory fast enough to keep up with the application's allocation requests.

There are two main approaches to schedule garbage collection work. One approach is work-based, i.e., for every allocation performed by the application, some garbage collection work will be performed in a way to ensure that sufficient GC progress is made to satisfy all allocation requests. Alternatively, time based GCs need to be scheduled such that the thread or threads assigned to perform garbage collection work receive enough CPU time to keep up with the worst case allocation rate of the application. Work based GCs are easier to configure since they do not require an analysis of the application's worst-case allocation rate, while time based GCs permit faster memory allocation since no GC work is required at allocation time.

The concurrent, parallel, real-time GC presented in this paper is intended to work independently of the garbage collection scheduling scheme: The GC work is split into very small incremental steps as required for a work based collector, but it is also possible to assign a subset of the CPUs to perform these steps in parallel to the application.

1.2 The JamaicaVM single-CPU real-time GC

The basic GC algorithm that is used in the single-CPU version of JamaicaVM is a simple incremental mark and sweep collector as described by Dijkstra et al. in 1976 [6]. This algorithm maintains three sets of objects that are distinguished by the colours *white*, *grey* and *black*. A GC cycle starts with all objects being *white*, and the objects reachable from root pointers are marked *grey*. It then proceeds with the mark phase as long as there are *grey* objects by taking a *grey* object *o*, marking all *white* objects referenced by *o grey*, and marking *o* itself *black*. A write-barrier ensures that modifications of the object graph performed by the application will not result in the GC missing reachable objects. When the *grey* set is empty, all *white* objects are known to be garbage, so their memory will be reclaimed in the following sweep phase. In this phase, *black* objects are converted back to *white*, so that the next cycle can start with all allocated objects being *white*.

1.2.1 Synchronisation points

Even though JamaicaVM maps Java threads 1:1 to native threads, it uses its own scheduler running on top of the (real-time) OS's scheduler [25]. This approach provides a finer-grained control over scheduling decisions within Java code. The single-CPU scheduler uses synchronisation points that are introduced by the VM and compilers within all methods and all loops. Even though Java threads use OS threads, thread switches between the threads of one VM are restricted to these synchronisation points. At most one thread is in state *RUNNING*, such that all actions in between two synchronisation points are automatically atomic.

This fact is used by the operations of the GC: write barrier code, single steps of GC work during mark and sweep phases, etc., all these code sequences are automatically atomic with respect to one another as long as they do not contain synchronisation points.

1.2.2 Using fixed-size blocks

The JamaicaVM GC does not deal with Java objects or Java arrays directly and does not know about their structure. Instead, it works on single fixed-size blocks[1]. Working on blocks simplifies the GC significantly, while it automatically provides small units of GC work that can be done in incremental steps: The basic operations performed by the GC are scanning or sweeping a single block. The GC's work may be pre-empted after each of these basic increments, so that other threads can run even while GC work is going on.

Java objects or arrays that do not fit into a single fixed-size block are represented as a graph of fixed-size blocks that may be non-contiguous in memory. The access times to objects and arrays is still bounded and efficient when using this object layout as described in [26]. This paper also analysed the memory loss due to internal fragmentation and compared it to different versions of JDK. The result is that overall memory demand depends on the kind of allocations performed by the application, the overall memory demand is sometimes better and sometimes worse compared with a defragmenting GC.

1.2.3 Representation of mark colour

JamaicaVM uses one word per block for the colour. For colours *white* and *black*, this word contains a special value (*0* and *-1*, respectively). Any other value indicates that the block is *grey*. All *grey* objects are stored in a linked list, using the colour word to store the reference to the next element in this list. Adding a block to and removing the first block from the grey-list are efficient operations that can be performed in constant time, so that a complete GC cycle is guaranteed to finish in a time that is linear in the number of allocated blocks.

[1] The typical size of these blocks is 32 bytes or eight 32-bit machine words.

1.2.4 Write-barrier code

The write-barrier of the single-CPU JamaicaVM GC ensures that the strong tricolour invariant

There are no pointers from a black block to a white block

always holds. There are two main situations that require execution of the write-barrier code to ensure that this invariant holds: first, at each store of a reference in a *black* block and second, during a mark step that requires that no blocks referenced from the scanned block are *white* when the scanned block's colour is changed to *black*.

For any write of a reference that refers to a *white* block into a block on the heap, the written reference is added to the list of *grey* blocks. Since the write-barrier shades an object from *white* to *grey*, this operation is called shading. The code to shade a reference is shown in Figure 1.

```
1: shade(ref r)
2: {
3:   if (r->colour == white)
4:   {
5:     r->colour = grey_list;
6:     grey_list = r;
7:   }
8: }
```

Figure 1. Single-CPU code to shade a *white* block

1.2.5 Constant time root scanning

Scanning the stacks to find references typically requires stopping the corresponding thread, which can cause pauses that are too long for time-critical applications.

The solution employed by JamaicaVM that was originally published in [27] is to ensure that all root references that exist have are present on the heap as well whenever the GC might become active. The compiler and VM generate additional code to store locally used references to a separate root array on the heap.

All references that have a life span during which the GC might become active or that contains a synchronization point need to be copied to the root array. The normal write-barrier code is used to store references in the root array. The fact that all CPUs will be forced to run over a synchronization point at the start of a GC cycles (see section 3.1) ensures that all locally held references will be stored into root arrays. In conjunction with the snapshot-at-beginning write barrier, this will ensure that all root references that exist at the beginning or that will be read into local variables later will be found reachable during the mark phase.

To ensure that the GC will find all root references, it is sufficient to have a single global root pointer that refers to a list of all root arrays. The root scanning phase at the beginning of a GC cycle can now be reduced to shading the single global root pointer. The GC will then eventually traverse all root arrays. Since all live references have been stored in the root arrays, all local references will be found by the GC.

1.2.6 Mark and sweep steps

With the elimination of an explicit root scanning phase and the lack of a need for a compaction phase due to the object model based on fixed-size blocks, the GC work is reduced to the mark and the sweep phase. These phases work on single blocks, i.e., one step in the mark phase means scanning the reference of one *grey* block and one step in the sweep phase means checking one block' colour and reclaiming its memory in case it is *white*.

Consequently, one GC cycle consists of very small and uniform incremental mark or sweep steps. The total number of mark and

sweep steps is limited by the amount of allocated memory (free memory ranges are ignored by the sweep phase).

2. Towards a Parallel Collector

The existing single-CPU GC has been extended to enable parallel and concurrent execution. This means, first, that several application threads may execute in parallel and perform memory related operations such as memory accesses, write barrier code or memory allocation in parallel. Also, it means that one or several threads may perform GC work in parallel to the mutator threads. The GC work must still be divisible into small and uniform steps to be able to measure the amount of GC work performed if such a collector is to be employed as a work based collector. Finally, the number of work steps required to finish a complete cycle must be limited.

The following subsections explain the basic infrastructure that was built for the parallel GC: the scheduler runs multiple threads in parallel and provides new synchronisation primitives (section 2.1); the representation of the marking colours *white*, *grey* and *black* of blocks is fundamental for the overall performance (section 2.2); the write barrier itself needs to be changed (sect. 2.3); and the representation of free lists is also crucial (2.4). With all these mechanisms in place, the actual implementation of the parallel GC is relatively simple and will be explained in the next section 3.

2.1 Scheduler and synchronisation primitives

As for the single-CPU VM, the parallel VM uses synchronisation points to restrict thread switches. However, the number of *RUNNING* threads has been increased to a run-time constant n, which must be less or equal to the number of logical hardware CPUs available on the system. Any thread that is not in state *RUNNING* is stopped at a synchronisation point, but threads that are in state *RUNNING* may execute completely in parallel.

For OSes that provide APIs to select CPU affinity, these APIs are used to enforce that each *RUNNING* thread runs on a distinct CPU. The underlying OS scheduler may, however, pre-empt a *RUNNING* Java thread to execute threads not attached to the VM.

2.1.1 CPU-local structures

Having control over the set of *RUNNING* threads makes it possible to assign a CPU-local data structure to each *RUNNING* thread. This CPU-local data structure can be accessed by the thread that is running on this CPU exclusively without synchronisation with other threads as long as no synchronisation point is executed.

Different parts of the parallel GC implementation make use of these CPU-local data structures. In particular there are CPU-local *grey* sets (see section 2.2.1), CPU-local free lists (section 2.4) and CPU-local sweep sets (section 3.3).

2.1.2 Compare-And-Swap for one-way state changes

```
 1: CAS(VAR var,expected,new)
 2: {
 3:   word result;
 4:   atomic
 5:     {
 6:       result = var;
 7:       if (result == expected)
 8:         {
 9:           var = new;
10:         }
11:     }
12:   return result;
13: }
```

Figure 2. Semantics of *CAS* as used in this paper

A compare-and-swap (CAS)[2] is an atomic operation with the semantics shown in Figure 2. An example of the use of a CAS for a one-way state change is the phase of the GC: a phase change from *MARK* to *SWEEP* may be attempted by several CPUs in parallel using compare-and-swap. Only one CPU will succeed with this operation. However, the failed CPUs do not need to retry the CAS since the operation failed because some other CPU made exactly the state change that the failed CPUs attempted to make. In contrast, a CAS in the general case can fail and require one or several retries in case another CPU in parallel writes the same word. One-way state changes such as the GC phase change do not require a retry even when performed in parallel.

Care is needed to ensure that a CAS on a state change does not fail due to a cycle of states (the ABA problem): e.g., imagine CPU1 attempts to change the GC state from *MARK* to *SWEEP* and this unlucky CPU is so slow (it may be pre-empted by the OS to perform some completely different task) that another CPU2 will perform the state change to *SWEEP*, will finish all the sweep phase, restart the next GC cycle such that we are back in *MARK* phase. If now CPU1 will execute its compare-and-swap, it will perform the state change to *SWEEP* even though it has not ensured that the mark phase of the new cycle has finished. To solve this, for any such cycles of states, it has to be ensured that at some point during the cycle, no CPU is in the middle of performing a state change. This will be ensured by the synchronisation enforced at the beginning of each GC cycle (see section 3.1).

In general, a CAS for a one-way state change can be used in a real-time system straightforwardly, a worst-case execution time can be found since no retries are needed.

2.1.3 Compare-And-Swap with retry

Less useful for real-time systems are CAS-loops that retry the CAS operation in case it failed. This approach is sufficient for non-real-time systems that are optimised for average throughput, but not being able to limit the number of CAS-retries is not acceptable in a real-time system.

The number of CAS-loop iterations can, however, be limited to n if the number of CPUs is limited by n and the number of CPU cycles spent in between two successive executions of such a loop is at least *n-1* times larger than the time required for one CAS-loop iteration. The reason for this is that if a CAS failed on one CPU, a competing CAS on another CPU must have been successful. Hence, this other CPU will then perform work outside the CAS-loop. The same will happen for each following iteration with a failing CAS. After *n-1* failed CAS-loops, all other CPUs will be in the CAS-free code section and the nth CAS is guaranteed to succeed[3].

In consequence, a CAS with a retry-loop is only usable in a real-time system if the CAS-free code section after the CAS-loop can be made large enough. Ideally, the length of the CAS-free part should be configurable by a run-time constant to permit scaling for arbitrary numbers of CPUs.

2.1.4 Waiting for CPU synchronisation

At certain points, the presented parallel GC requires to ensure that no other CPU is in the middle of some activity, e.g. in the middle of performing write-barrier code or in the middle of scanning a *grey* object in a mark step.

[2] For simplicity, this paper ignores that the cost of a CAS may grow with the number of CPUs accessing the same same address simultaneously.

[3] On real systems, determining the precise time spent in the loop is more complex, including interrupts or cache effects. Nevertheless, using a large enough code sequence between two successive CAS-loops can be used to reduce the probability of exceeding n iterations to an arbitrarily low value.

The use of synchronisation points provides a means to get notified when a given CPU has finished all the activity in between two synchronisation points. The parallel JamaicaVM scheduler provides a function *waitForSync* that halts the current CPU (A) until another CPU (B) has reached its next synchronisation point. The implementation is fairly simple, a new state *SYNC* is used to implement this function as shown in Figure 3: The state of CPU B is set to SYNC and then a busy waiting loop waits for CPU B to reset its state[4]. Resetting the state is performed at each synchronisation point, the code for such a synchronisation point is shown in Figure 4: In case the state is not *RUNNING*, a single compare-and-swap will be used to reset the state from *SYNC* to *RUNNING*.

The actual implementation also provides a mechanism to avoid a live-lock in case threads are waiting for one another's synchronization. In case *waitForSync* is performed to wait for a CPU that itself performs *waitForSync*, this call will return immediately.

```
1: waitForSync(Cpu otherCpu)
2: {
3:   CAS(otherCpu->state,RUNNING,SYNC);
4:   while (otherCpu->state == SYNC)
5:     { /* busy wait loop */
6:     }
7: }
```

Figure 3. Simplified implementation of *waitForSync*

```
1: if (cpu->state != RUNNING)
2: {
3:   if (CAS(cpu->state,SYNC,RUNNING) != SYNC)
4:     {
5:       [..]
6:     }
7: }
```

Figure 4. Code for a sync point for a thread running on *cpu*

For *waitForSync* to be usable in the parallel real-time GC, an upper bound for the time to perform a *waitForSync* is required. The VM and compilers ensure that the length of code executed in between two synchronisation points is restricted by t_{sync}. Together with the time $t_{cas,while}$ required for the CAS and one execution of the while-statement, the worst-case execution time t_s for *waitForSync* can be determined (in the following, *wcet(f)* will be used to as the worst-case execution time of a function *f*):

$$t_s := t_{sync} + t_{cas,while} \quad (1)$$

2.2 Colour encoding

Any block on the heap needs to be marked with a marking colour, *white*, *grey*, or *black*, such that all blocks are grouped in three disjoint sets of *white*, *grey*, and *black* blocks. Important operations on blocks are shading of *white* blocks (as required by the write-barrier, see section 2.3 or during the mark phase, section 3.2), obtaining one *grey* block from the grey set (required to perform a mark step on this block), and determining that the *grey* set is empty (which means that the mark phase is finished).

Similar to the single-CPU GC, one word per block is reserved for the colour such that *grey* blocks can be stored in a singly linked list. For parallel mutators and a parallel GC, using only one single list for the *grey* set, however, would result in an important bottleneck since write barrier code and mark phase work would access this list in parallel.

2.2.1 CPU-local *grey* lists

Instead, it was chosen to maintain several CPU-local linked list to each represent a subset of the *grey* blocks. Since each CPU is allowed to access only its local *grey* list, no synchronisation is required here. However, it has to be ensured that several CPUs that shade a block in parallel will not result in an inconsistent state.

2.2.2 Shading blocks

The shading code is shown in Figure 5. A compare-and-swap is used to change the colour of an object referenced by *r* from *white* to the address of the first element in the current CPU's *grey* list. If this CAS was successful, the grey list of the current CPU will be set to *r*, which becomes the new head of this list.

```
1: shade(Cpu cpu, ref r)
2: {
3:   if (CAS(r->colour,white,cpu->greyList) == white)
4:     {
5:       cpu->greyList = r;
6:     }
7: }
```

Figure 5. Code to shade a *white* block in the parallel GC

If the shading code is performed in parallel by several threads, only one of these threads' CAS will be successful. Then, the block has been shaded by one thread, so it is not *white* any more and hence it does not need to be shaded by any other thread. The CAS performs a one-way state change that does not require a retry.

2.2.3 Obtaining *grey* blocks

Finding an element in the *grey* set is an important operation during the mark phase (see section 3.2). With CPU-local *grey* lists, this operation is very simple as long as the CPU-local list is not empty. The code is shown in Figure 6[5]. A new colour *anthracite* is introduced for *grey* blocks that are currently scanned by one CPU performing a mark step. Therefore, the head of the CPU's local *grey* list is taken and, if no other CPU is performing or has performed ((i.e., it is *anthracite*) or *black*, resp.) a mark step on this block, it will be marked *anthracite* to indicate that it will be marked by the current CPU and the CPU's *grey* list will be set to the tail of the list.

In case *getGrey* failed to find a *grey* block in the head of the current CPU's *grey* list or it failed to set its colour to *anthracite*, the current CPU's *grey* list is empty. In this case, however, it might be possible to find a *grey* block in the *grey* list of a different CPU, so *getGrey* attempts to *steal* a *grey* block from a different CPU.

Stealing a set of *grey* block from a different CPU is performed by changing the colour of the head of the other CPU's *grey* list to *anthracite*. The code is shown in Figure 7: This routine attempts to change the colour of the head of the *grey* set of another CPU to *anthracite*. In case it is successful, all elements of the list are taken from the other CPU, i.e., the tail of the list is stored as the grey list of the current CPU. This is required since the *greyList* is a CPU-local field and we may not modify another CPU's local fields.

In case *steal* is not successful, *NULL* will be returned. This, however, does not mean that the mark phase can finish: other CPUs might still be active performing mark steps or write-barrier code and adding new *grey* blocks to their *greyList*s.

[4] The real code is somewhat more complex since it has to deal with the case that CPUs might have been pre-empted by the OS to execute a different thread outside of the VM. In this case, the pre-empted thread will inherit the priority from the current thread and, if the busy loop is not successful after a few iterations, a blocking wait will be performed.

[5] The actual code shown here is simplified, the actual implementation is more complex since it deals with subtle race conditions still present in the pseude-code shown here.

```
 1: ref getGrey(Cpu cpu)
 2: {
 3:   ref r = getAnthracite(cpu,cpu->greyList);
 4:   if (r == LAST_GREY)
 5:     {
 6:       r = steal(cpu);
 7:     }
 8:   return r;
 9: }
10:
11: /* helper to mark r anhtracite */
12 ref getAnthracite(Cpu cpu, ref r)
13: {
14:   if (r != LAST_GREY)
15:     {
16:       ref c = r->colour;
17:       if ((c != ANTHRACITE) &&
18:           (c != BLACK     ) &&
19:           CAS(r->colour,c,ANTHRACITE) == c)
20:         {
21:           cpu->greyList = c;
22:           return r;
23:         }
24:     }
25:   return LAST_GREY;
26: }
```

Figure 6. Routine to obtain a *grey* block and mark in *anthracite*.

```
 1: ref steal(Cpu cpu)
 2: {
 3:   for (Cpu otherCpu : allCpus \ { cpu } )
 4:     {
 5:       ref r = getAnthracite(cpu,ohterCpu->greyList);
 6:       if (r != LAST_GREY)
 7:         {
 8:           return r;
 9:         }
10:     }
11:   return NULL;
12: }
```

Figure 7. Routine to attempt to steal a *grey* block from a different CPU's *grey* list and mark it *anthracite*.

This work stealing mechanism is as coarse as it could be: If a CPU manages to steal *grey* blocks from another CPU, it will steal all *grey* blocks from that CPU. This behaviour is ideal for the case that the other CPU does not perform GC work, but only mutator work that adds elements to its grey list when executing write barriers. However, if all CPUs involved perform GC work, this may degrade to all CPUs competing for a single remaining list of *grey* blocks. Current measurements show that this does not seem to occur often in practice, but this point should be analysed further in future work.

2.3 Write-barrier code

The single-CPU collector uses an incremental-update write-barrier, i.e., whenever a reference to a *white* object *a* is stored into another object, *a* is added to the *grey* set [19]. This write barrier has the advantage that memory that becomes unreachable during a GC cycle might be reclaimed during this cycle, such that memory may be reclaimed early and less memory needs to be marked.

However, this approach is not feasible when several mutators run in parallel. One mutator *m1* might read a reference to an object *b* from the heap, and another mutator *m2* might overwrite this reference before *b* was added to the *grey* set. Then, *b* could only be found to be reachable from the roots of *m1* by re-scanning these roots. Rescanning the roots at the end of the mark phase, however, makes termination of the mark phase difficult.

Therefore, the incremental-update write-barrier was replaced by a snapshot-at-beginning [31] write-barrier that shades deleted references that refer to *white* objects. This write-barrier ensures the weak tricolour invariant (see [19]):

> *All white objects pointed to by a black object are reachable from some grey object through a chain of white objects*

Using this snapshot-at-beginning barrier ensures that all memory reachable at the beginning of a GC cycle and all memory allocated during the cycle will survive the current GC cycle. Compared to the original approach, some objects may therefore be reclaimed later, but the worst-case behaviour is the same since the incremental-update write-barrier gives no guarantee that objects that become unreachable will be reclaimed during the current cycle.

2.4 Free lists

A central aspect of a parallel GC and a parallel memory allocation routine is the data structure used to store free memory. Since JamaicaVM uses graphs of fixed-size blocks to represent objects of arbitrary sizes, there is no need to distinguish different size classes of free objects. An efficient data structure to hold the set of available fixed-size blocks is sufficient.

The operations required on the set of free blocks are allocation of a single block by a mutator thread and the addition of a block during the GC's sweep phase (see section 3.3). It has to be possible for several threads running on different CPUs to perform these operations in parallel.

To enable parallel access, CPU-local free lists are used. Allocating a block from the CPU-local free list or adding a block to this list can be performed without any synchronisation with other CPUs. The maximum number of blocks stored in a CPU-local free list is limited by a constant *MAX_FREELIST_SIZE*[6]. Whenever a block is allocated and the CPU-local free list is empty, or when a block is freed and the CPU-local free list contains this maximum number of elements, one set of such objects is filled from or spilled to a global list[7]. This global list uses synchronisation via compare-and-swap and a retry in case the operation failed. As an example, the code to allocate one block is shown in Figure 8.

Allocation of one block usually means just unlinking the first block from the CPU-local free list. Only if this list is empty, one new set of blocks will be taken from the global free list using a compare-and-swap. This compare-and-swap might fail if another CPU modifies this global list in parallel, so a retry is performed (see 2.1.3).

The number of retries is bounded if *MAX_FREELIST_SIZE* is set to a value large enough for the number of CPUs *n*. Determining a value that is large enough means analysing the time required for one iteration of the CAS-loop t_l and comparing this to the time t_a required for a call to *alloc* that can be satisfied from the CPU-local free list. *MAX_FREELIST_SIZE* must be set such that

$$t_a \cdot \mathit{MAX\_FREELIST\_SIZE} \geq \mathrm{n} \cdot \mathrm{t_l} \qquad (2)$$

Our implementation also measures the maximum number of iterations required for every CAS-loop, such that a value that is too low will eventually be detected at run-time.

The allocation colour is always *black*, i.e., a newly allocated block will not be freed in the current GC cycle, which would be

[6] In the current implementation, this constant is set to 64 blocks.

[7] Repeated freeing and allocating of one block by one CPU could cause repeated fills for an empty CPU-local set followed by spills of the just filled set. To avoid this, two CPU-local free lists are used, one only for freeing and the other one for allocating.

```
 1: ref alloc(Cpu cpu)
 2: {
 3:   ref r = cpu->freeList;
 4:   if (r == NULL)
 5:     { /* fill from globalFreeList: */
 6:       do
 7:         {
 8:           r = globalFreeList;
 9:           if (r == NULL)
10:             { /* out of memory */
11:               return NULL;
12:             }
13:         }
14:       while (CAS(globalFreeList,r,r->nextSet) != r);
15:     }
16:   cpu->freeList = r->next;
17:   r->color = BLACK;
18:   return r;
19: }
```

Figure 8. Routine to allocate one block from a CPU-local free list and fill this list from the global list in case it is empty.

against the snapshot-at-beginning principle. However, if a newly allocated block becomes unreachable before the end of the current GC cycle, it is guaranteed to be collected during the next cycle.

3. Phases of the GC Cycle

This section describes the phases of one cycle of the parallel real-time GC. All CPUs that perform GC work are always in the same phase; the value of a global variable *gcPhase* gives the current phase. A GC cycle always starts with a *FLIP*, which is followed by the *MARK* and *SWEEP* phases. For Java, a *FINALISATION* phase is also required to take care of Java objects with finalizers.

It has to be ensured that several CPUs can perform single steps simultaneously in each phase, but it also has to be ensured that phase changes will be performed properly even if several CPUs simultaneously detect that the current GC phase is finished and a new phase should start. Therefore, compare-and-swap is used to perform the phase changes. Only one CPU will be successful in performing the phase change, any other CPU that is not successful performing a phase change can continue with the GC work for the new phase, no retry is required.

3.1 GC cycle start: FLIP

The start of each new GC cycle happens after the last cycle's sweep phase has finished. So, a thread that has detected that the sweep phase has finished (see section 3.3) will initiate the start of a new cycle. The initial GC phase is called *FLIP* and is a very short operation that is performed by the first CPU that succeeds on the CAS that changes the phase from *SWEEP* to *FLIP*.

At the beginning of a GC phase, all allocated objects are marked *black*, there are no *grey* or *white* objects. The single root object will be marked *grey* and a 'flip' will be performed, i.e., the meanings of *black* and *white* will be exchanged. However, before the mark phase can start, it has to be ensured that all CPUs are aware of the fact that the meanings of *black* and *white* have been exchanged. Therefore, the thread that performs the flip will wait for all other CPUs to synchronise using the *waitForSync* operation. Only then the mark phase will start.

In case several CPUs detect that a GC phase has finished simultaneously, all others that are not successful on the CAS that changes the GC phase from *SWEEP* to *FLIP* will *waitForSync* on the CPU that was successful.

The total time spent for the *FLIP* is equal to the time required for the flip code plus the time for all other CPUs to run to the next sync point. I.e., the worst-case execution time is *wcet(flip)* + t_s.

The worst-case CPU time occurs if $n-1$ CPUs attempt to start a new GC cycle, such that $n-2$ CPUs stall waiting for the flip to complete, while the flipping CPU has to *waitForSync* on the last remaining CPU executing application code. Hence, the worst-case CPU time t_{flip} required is limited as follows.

$$t_{flip} \leq wcet(flip) + (n-1) \cdot t_s \tag{3}$$

Apart from the *FLIP* phase, no explicit root scanning phase is required since JamaicaVM and its compilers ensure that all locally used references are also stored in the heap at every sync point. Since the *FLIP* requires all CPUs to execute one sync point, it will be ensured that all locally used references will have been stored in the heap before the *FLIP* is finished.

3.2 Mark phase

One step during the mark phase is simple: A *grey* block *b* needs to be obtained using the *getGrey* function described above (in section 2.2.3, Figure 5). Then, all blocks referenced from *b* that are still *white* must be shaded using the *shade* function (see section 2.2.2, Figure 6) and *b* itself must be marked *black*.

Problematic is the case if *getGrey* fails to return a *grey* block. This might mean either that the *grey* set is empty and the mark phase is finished, or it might mean that there are temporarily not enough elements in the *grey* set to allow all CPUs that are assigned to perform GC work to scan blocks in parallel. Earlier research has found an upper bound for the frequency of the case that the *grey* set will have too few elements [28]. The current CPU can make no progress in this case, it can only stall until the *grey* set contains sufficient elements again. The number of these stalls during the mark phase is limited by $(n-1) \cdot d$ on a system with n CPUs performing mark work in parallel and for a heap depth of d [28]. The heap depth d is the maximum distance of any reachable block on the heap from the root reference of the heap graph. d depends on the application, a parallel mark phase is possible only for mutators that create a heap graph with $d << heapsize$.

It is important to distinguish the case of a stall from the case that the mark phase is finished. Therefore, after each stall, it is checked whether any other CPU performed any shading. If no other CPU shaded any new blocks, then the *grey* set must be empty and the mark phase is finished.

The code for a mark step is shown in Figure 9: A *grey* block *b* is obtained and all blocks referenced by *b* are shaded before *b* itself is marked *black*. If no *grey* block was found, a stall is performed via a call to *waitForSync* for all CPUs. In between two mark steps, a sync point is executed, such that *waitForSync* will return after a mark step. Finally, if no CPU made any mark progress since the last stall, the mark phase is finished and the SWEEP phase starts.

The time required for a normal mark step is limited and fairly uniform since each mark step marks the references in a single fixed-size block (which has 32 bytes for the JamaicaVM implementation). The worst-case execution time for this is *wcet(*$mark_{nostall}$*)*. Any allocated block will be marked only once, so for *a* allocated blocks, the total marking time will be $a \cdot$ *wcet(*$mark_{nostall}$*)*.

The time required for a stalling mark step is t_s and the number of stalls is limited by $(n-1) \cdot d$, such that the total time spent for stalls is limited by $(n-1) \cdot d \cdot t_s$.

In total, the CPU time t_{mark} spent for mark steps is limited:

$$t_{mark} \leq a \cdot wcet(mark_{nostall}) + (n-1) \cdot d \cdot t_s \tag{4}$$

3.3 Sweep phase

The purpose of the sweep phase is to add all blocks that remained *white* after the mark phase to the free lists. To enable parallel

```
 1: void mark(Cpu cpu)
 2: {
 3:   ref b = getGrey(cpu);
 4:   if (b != NULL)
 5:     {
 6:       for (ref r in *b)
 7:         {
 8:           shade(r);
 9:         }
10:       b->colour = BLACK;
11:     }
12:   else
13:     { /* grey set empty: stall (actual implementation
14:          does not need to iterate over all CPUs) */
15:       for (Cpu otherCpu : allCpus \ { cpu } )
16:         {
17:           waitForSync(otherCpu);
18:         }
19:       if (no CPU performed mark work)
20:         { /* mark phase is finished */
21:           CAS(gcPhase,MARK,SWEEP);
22:         }
23:     }
24: }
```

Figure 9. Routine to perform one step in the GC mark phase.

sweeping, the heap is partitioned into *sweep sets*[8] of s contiguous blocks. A CPU that performs sweep work will exclusively work on its CPU-local sweep set, such that no particular synchronisation with other CPUs that work on different sweep sets is required.

Only when the CPU-local sweep set is empty, the next set will be taken from a global pool of memory ranges that need to be swept. Accesses to this global pool are performed using compare-and-swap, and a retry is required in case the compare-and-swap failed. The number of retries on an n processor system, however, is limited by $n-1$ if the sweep set size s is large enough (see 2.1.3).

The end of the sweep phase is reached when the global pool of sweep sets is exhausted. However, at this point, some CPUs might still perform sweep steps on their CPU-local sweep sets. The GC phase cannot terminate before all CPU-local sweep sets have been fully processed. Therefore, at this last phase of the sweep phase, any CPU that has to perform GC work but has an empty local sweep set will perform sweep steps for other CPUs that are not done with their CPU-local sweep set yet. The code to increment the position in the current sweep set uses a compare-and-swap operation, but no retry is required since the counter can only increase and a failed compare-and-swap means that some other CPU has increased the counter already (see section 2.1.2).

The code of the parallel sweep phase is shown in Figure 10.

Since the whole heap needs to be scanned during sweep, the total number of sweep steps that are required is the number of blocks in the heap h. Since the last sweep set is scanned in parallel by up to n CPUs, in the worst case, we will have $(n-1)\cdot s$ additional attempts to sweep in parallel.

In total, the CPU time t_{sweep} spent for sweep is limited by:

$$t_{sweep} \leq wcet(sweep) \cdot (h + (n-1)\cdot s) \tag{5}$$

3.4 Finalisation phase

The finalisation phase is required in a GC for Java to ensure that unreachable objects that define a *finalize* method will be scheduled for finalisation. The JamaicaVM GC dedicates a specific phase after the mark phase to finalisation The single-CPU GC keeps a doubly-linked list of objects that needed finalisation and, during the finalisation phase, the GC iterates over the list. For every node

[8] In the current implementation, one sweep set has 128 blocks.

```
 1: void sweep(Cpu cpu)
 2: {
 3:   Set sweepSet = cpu->sweepSet;
 4:   ref r = sweepSet.removeBlock();
 5:   if (r == NULL)
 6:     { /* get new local sweep set */
 7:       sweepSet = globalSweepSets.getNext();
 8:       if (sweepSet != NULL)
 9:         { /* use new local sweep set */
10:           r = sweepSet.removeBlock();
11:         }
12:       else
13:         { /* no more sweep sets, use other CPU's */
14:           for (Cpu otherCpu : allCpus \ cpu)
15:             {
16:               sweepSet = otherCpu.sweepSep();
17:               r = sweepSet.removeBlock();
18:               if (r != NULL)
19:                 {
20:                   break;
21:                 }
22:             }
23:           if (r == NULL)
24:             { /* sweep phase is finished */
25:               CAS(gcPhase,SWEEP,FLIP);
26:               /* flip will waitForSync to ensure
27:                  all sweep steps are completed */
28:               return;
29:             }
30:         }
31:       cpu->sweepSet = sweepSet;
32:     }
33:   /* use CAS in case 2 CPUs sweep the same block */
34:   if (CAS(r->colour,WHITE,FREE) == WHITE)
35:     {
36:       free(r);
37:     }
38: }
```

Figure 10. Routine to perform one step in the GC sweep phase.

with an object that became unreachable, that object will be added to a list of finalisable objects that are dealt with by the finalizer thread.

The parallel GC cannot simply iterate and modify such a list. Instead, each CPU that performs finalisation work will obtain exclusive access to on element in the list using a single cas. If the CAS was successful, the object will be added the list of reachable objects that will be checked again in the next GC or the list of objects that require finalisation.

If a CPU that wants to perform finalisation work fails to obtain this exclusive acces using a single CAS, this CPU will continues without doing any GC work. This is possible since the number of elements in the finalisation list is typically small. If this number is limited by a constant f for an application running on n CPUs, then the number of finalization steps that may make no progress is limited by $d \leq f \cdot (n-1)$.

In total, the CPU time $t_{finalize}$ spent for finalisation is limited by the number of objects that require finalisation f and the worst-case execution time of one finalisation step plus the time spent for failed finalization steps:

$$\begin{aligned} t_{finalize} &\leq f \cdot wcet(final) + f \cdot (n-1) \cdot wcet(final) \\ &= n \cdot f \cdot wcet(final) \end{aligned} \tag{6}$$

4. Analysis and Measurements

This section first gives a theoretical analysis to show that the amount of work to finish a full GC cycle is bounded. The second part of this section then presents measurements made with the implementation.

4.1 Work required for GC cycle

With the time required for the GC phases as shown in equations (3) through (6), it is now easy to sum up the total worst-case execution time t_{gc} required for a whole GC cycle as

$$\begin{aligned} t_{gc} &\leq t_{flip} + t_{mark} + t_{sweep} + t_{finalize} \\ &= \textit{wcet(flip)} + (\mathrm{n}-1)\cdot \mathrm{t_s} \\ &\quad + \textit{wcet(mark}_{nostall}\textit{)} \cdot \mathrm{a} + (\mathrm{n}-1)\cdot \mathrm{d}\cdot \mathrm{t_s} \\ &\quad + \textit{wcet(sweep)} \cdot (\mathrm{h} + (\mathrm{n}-1)\cdot \mathrm{s}) \\ &\quad + \textit{wcet(final)} \cdot \mathrm{n}\cdot \mathrm{f} \end{aligned} \tag{7}$$

If we define the time t_{step} for one step of GC work as the maximum time required for one step during phases flip, mark, sweep, or finalisation

$$\begin{aligned} t_{step} = max \quad \{ \quad & \textit{wcet(flip)}, \\ & \textit{wcet(mark}_{nostall}\textit{)}, \\ & \textit{wcet(sweep)}, \\ & \textit{wcet(final)}, \\ & t_s \} \end{aligned} \tag{8}$$

and with the knowledge that the allocated number of blocks a is always less than the total number of blocks h, $a \leq h$, we can simplify (7) to

$$t_{gc} \leq t_{step} \cdot (2 \cdot h + n + (n-1)\cdot (d+s) + n \cdot f) \tag{9}$$

With the application and configuration-dependent constant

$$c = n + (n-1)\cdot (d+s) + n \cdot f \tag{10}$$

which depends on the number of processors n, the sweep set size s, the maximum depth of the object graph d[9] and the maximum length of the finalisation list f, we get the simplified form

$$t_{gc} \leq t_{step} \cdot (2 \cdot h + c) =: t_{gcmax} \tag{11}$$

Hence, the total number of steps of GC work that need to be performed to complete a full GC cycle is limited by the runtime constant t_{gcmax}. For any application with limited memory demand, a work based scheme can use this fact to perform enough GC work on each allocation to ensure that the GC finishes its cycle and reclaims enough memory to satisfy all allocation requests.

In practice, the dominant part of t_{gcmax} is $t_{step} \cdot 2 \cdot h$, which is linear in the heap size, similar to existing single CPU collectors. For systems with many CPUS (> 128), a large depth of the object graph or large numbers of objects with finalizers (both $>10\%$ of the heap size), the factor proportional to c starts to dominate.

The author is not aware of any other concurrent and parallel GC that provides a similarly tight bound on the amount of GC work that guarantees finishing a complete cycle.

4.2 Measurements of low-level operations

Important for the validity of the worst-case execution time analysis is the fact that the worst-case number of loop iterations performed is actually limited as predicted. An allocation-intensive benchmark with eight parallel threads performing allocations of objects and arrays with different sizes and different lifespans was used. With the exception of the measurement of allocation times (section 4.2.2), the GC is run in work-based mode. The test application was run for 1Mio seconds (about 12 days) and the number of iterations in CAS-retry and in *waitForSync* loops was measured. The following sections present the results obtained on an eight logical processor Intel Xeon CPU E5405 machine running at 2.0GHz. The heap size was set to 32MB for this benchmark.

4.2.1 CAS-retry iterations

All CAS-retry loops of the parallel GC implementation were equipped with a counter that counts the number of retries that were actually performed. The results are shown in Figure 11.

As predicted, the number of tries for the CAS-loops on allocation, free, access to the global sweep set and changes to the global available memory counter are limited by eight on this eight-processor machine since after seven failing CAS, the other CPUs must all have performed a successful CAS and consequently will leave enough time for the failing CPU to be successful on the next try. The frequency of a retry is about 30 times lower than that of a successful CAS.

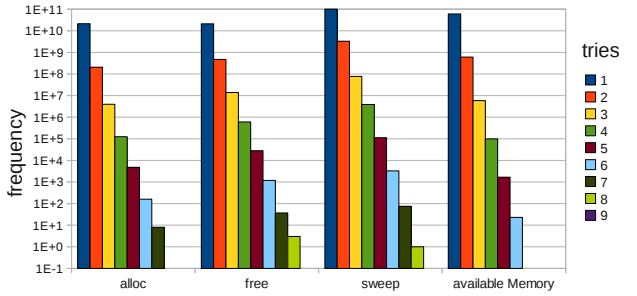

Figure 11. Number of CAS tries required on an allocation intensive benchmark.

4.2.2 Allocation times

A modified version of the allocation benchmark was used to measure the allocation time of allocating a small object (32 bytes). For this test, the application was run with 2 allocating threads in parallel and a background GC thread, using only 4 CPUs of the 8 CPU system[10]. The OS was Linux (2.6.18, CentOS), scheduling was set to SCHED_RR and the Linux priorities of the allocating threads were set to +19, the priority of the GC thread was set to +1. The measurements of the allocation thread that had the highest maximum allocation time are shown in Figure 12. The by far largest number of allocations require less than 1μsec, but some allocations require longer, up to 64μsec. These results are not satisfactory yet, the exact reasons for the outliers have to be identified. Similar outliers occur in single threaded applications when measuring other activities than allocatin, which means that at least some of these outliers are unrelated to the presented GC.

4.3 Measurements of scalability

To measure the scalability of the implementation using real-world applications, a parallel version of the SpecJVM98 benchmark suite [29] was created. This parallel version starts eight threads that repeatedly execute the eight benchmarks SpecJVM98 consists of in parallel. For each benchmark, the average execution time is measured. The parallel VM was run using 1, 2, 4, 6 and 8 of the available 8 logical CPUs. The results are shown in Figure 13.

[9] Without knowledge of the application, the maximum depth of the object graph is bounded only be the amount of reachable memory. Consequently, a good upper bound is only possible for applications with a proven and low limit for the depth of the object graph.

[10] The fourth CPUs is required such that service threads, e.g., the finalizer thread, can run without disturbing the other threads.

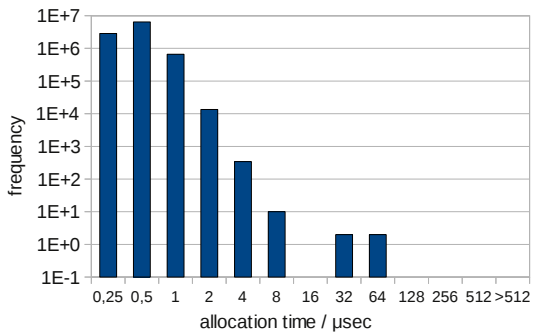

Figure 12. Frequencies of times required for allocation of a small object.

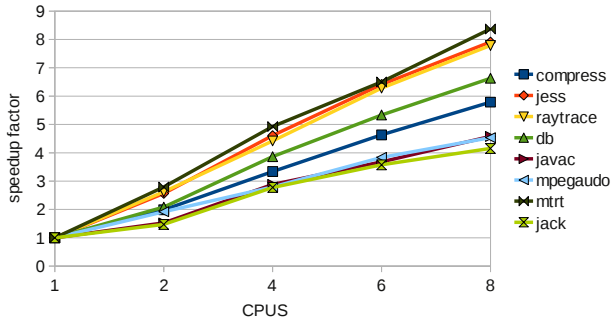

Figure 13. Scaling of the performance

The performance scales well with the number of CPUs: for 8 CPUs, a performance factor between 4 and 8 was measured. One benchmark (mtrt) even exceeds the factor 8, which is probably due to the fact that this is the only benchmark that is multi-threaded itself, so the threads of this benchmark may be priviledged by the scheduler when more CPUs are available.

5. Related Work

Early work on concurrent and incremental mark and sweep garbage collection dates back to the seventies with important contributions from Dijkstra [6] and Steele [30]. The first incremental copying collector was presented by Baker [3].

One of the earliest implementations of an incremental and parallel GC is the Multilisp implementation by Halstead [14]. This is a parallel version of Baker's copying garbage collector that uses processor-local old-spaces and new-spaces to enable parallel garbage collection work and parallel memory allocation. However, this approach did not address the problem of load balancing at all, while I assume perfect load balancing in this paper, i.e., as long as the *grey* set is non-empty, any processor could obtain an element from this set to perform mark phase work. Load balancing between different mutator processors was proposed by Endo [11]. In Endo's approach, each process maintains a local subset of the *grey* set. These local subsets are divided into two subsets: a local mark stack and a stealable mark set. When a processor would stall due to empty local mark sets, it will attempt to steal *grey* objects from another processor's stealable mark set.

Attanasio et al. made a comparison of the scalability of different parallel GC implementations (generational, non-generational, mark-and-sweep, semi-space copying) [1].

Endo, Taura and Yonezawa [12] presented an approach to predict the scalability of parallel garbage collection in the average case. Their approach takes the memory graph and specifics of the hardware such as cache misses into account. Their result is an estimate of the scalability, while this paper presents an upper bound for the worst-case scalability.

Blelloch and Cheng have presented a theoretical bound on time and space for parallel garbage collection [4] and refined their approach [5] to become practically implementable by removing the fine granularity of the original algorithm and adding support for stack scanning, etc. The original scanning of fields one at a time prevented parallel execution. Their new approach scans one object at a time, resulting in parallel scanning of objects. Since scanning of large objects would prevent parallelism, the authors split up larger objects into segments that can be scanned in parallel.

A parallel, non-concurrent GC with load balancing via *work stealing* was presented by Flood et al. [13]. Each processor has a fixed-size work-stealing queue. If a processor's work-stealing queue overflows, part of its content is dumped to a global overflow set. Processors with an empty local queue first try to obtain work from the overflow set before they resort to stealing. This technique was then applied to parallel copying and mark-compact GCs resulting in a speedup factor between 4 and 5 on an 8 processor machine.

The parallel, incremental and concurrent GC presented by Ossia et al. [18] employs a low-overhead work packet mechanism to ensure load balancing for parallel marking. In contrast to previous balancing work, all processors compete fairly for the marking work, i.e., there is no preference for a processor to first work on the work packets it generated.

Other fully concurrent on-the-fly mark-and-sweep collectors have been presented by Doligez and Leroy [8], Doligez and Gonthier [7], and Domani et al. [9, 10].

The possibility to implement the *grey* set by reserving one word per object such that all *grey* objects could be stored in a linked list as published earlier [20, 25].

A different approach to parallel garbage collection was presented by Huelsbergen and Winterbottom [16]. Their approach is basically a concurrent mark-and-sweep GC in which the mark and the sweep phases run in parallel. However, the mark phase itself is not parallel in this approach, so it does not suffer from the stalls discussed in this paper.

Being able to give an upper bound of the scalability of the garbage collector enables one to give an upper bound on the total work required to perform one garbage collection cycle. This work can then be used to schedule the GC such that it reclaims memory fast enough to satisfy all allocation requests. There are two main approaches to schedule the GC work: work based scheduling [3, 24], or time based approaches [2].

Performing GC work in a work based scheme or a time based scheme based on the allocation rate of an application was presented by Baker [3] and Henriksson [15], respectively. Schberl schedules the GC as an ordinary application thread [22].

Pizlo et al. recently presented different approaches for concurrent defragmenting real-time GCs [21]: Their CHICKEN collector aborts moving an object in case of a concurrent modification, while their CLOVER collector detects writes by using a marker value for fields of objects that are obsolete since they have moved. Their fine-grained synchronization results in μsec response times.

A non-blocking real-time GC implemented in hardware was presented recently by Schberl and Puffitsch [23].

6. Conclusion

The presented real-time GC can be used to run in parallel by an arbitrary number of CPUs and concurrently to an arbitrary number of application threads. The synchronisation required between

the threads involved is limited and the time for all operations that require synchronisation is bounded by a very short upper bound. The most frequent synchronisation operation used is CAS for one-way state changes or CAS-loops with bounded number of iterations. Less frequently, CPUs need to synchronise by waiting for the execution of the next synchronisation point, but even for this operation the execution time is bounded.

Furthermore, the GC work is performed in very small single steps, and the total number of these steps is linear in the heap size plus a run-time constant that depends on the number of CPUs, the depth of the heap graph and the number of objects requiring finalisation. The GC may consequently be used either as a work-based collector that performs GC work at allocation time, or as a time based collector.

The runtime overhead imposed on the application compared to the original single-CPU GC is very low. Measurements show a good scalability.

Acknowledgments

This work was partially funded by the European Commission's 7th framework program's JEOPARD project, #216682.

References

[1] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collectors. In H. G. Dietz, editor, *Proceedings of the Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science*, pages 177–192, Cumberland Falls, Kentucky, August 2001. Springer-Verlag.

[2] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor 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, January 2003. ACM Press.

[3] Henry 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] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

[5] Perry Cheng and Guy E. Blelloch. A parallel, real-time garbage collector. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 125–136, New York, NY, USA, 2001. ACM.

[6] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Lecture Notes in Computer Science, No. 46*. Springer-Verlag, New York, 1976.

[7] Damien Doligez and Georges 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. ACM Press, January 1994.

[8] Damien Doligez and Xavier 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, January 1993.

[9] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[10] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. Technical Report 88.385, IBM Haifa Research Laboratory, 2000. Fuller version of [9].

[11] Toshio Endo. A scalable mark-sweep garbage collector on large-scale shared-memory machines. Master's thesis, University of Tokyo, February 1998.

[12] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. Predicting scalability of parallel garbage collectors on shared memory multiprocessors. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA*, pages 43–43. IEEE Computer Society, 2001.

[13] Christine Flood, Dave Detlefs, Nir Shavit, and Catherine Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[14] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

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

[16] Lorenz Huelsbergen and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. In Jones [17], pages 166–175.

[17] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[18] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 129–140, Berlin, June 2002. ACM Press.

[19] Pekka P. Pirinen. Barrier techniques for incremental tracing. In Jones [17], pages 20–25.

[20] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07: Proceedings of the 6th international symposium on Memory management*, pages 159–172, New York, NY, USA, 2007. ACM.

[21] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 33–44, New York, NY, USA, 2008. ACM.

[22] 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. IEEE, 2006.

[23] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking real-time garbage collection. *Trans. on Embedded Computing Sys.*, accepted for publication, 2010.

[24] Fridtjof Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In Jones [17], pages 130–137.

[25] Fridtjof Siebert. Hard real-time garbage collection in the Jamaica Virtual Machine. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.

[26] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.

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

[28] Fridtjof Siebert. Limits of parallel marking garbage collection. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 21–29, New York, NY, USA, 2008. ACM.

[29] Standard Performance Evaluation Corporation (SPEC). *SPECjvm98 Benchmarks*. http://www.specbench.org/osg/jvm98/.

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

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