A hardware-assisted concurrent & parallel garbage collection algorithm

Greg Wright

November 10, 2008

Draft report - Please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

1. Introduction

The goal of the Maxwell project in Sun Labs is to support very large heaps (on the order of 512GB) and large numbers of threads efficiently in applications built for the Java™ platform, and we allow ourselves to make changes in the processor and memory system to support the Java virtual machine. The main challenge with such large heaps is garbage collection behavior.

We assume that the reader is familiar with the general idea of garbage collection, but start with an explanation of the problems faced when scaling to large heaps and thread counts. For more background we recommend the classic GC reference text [Jones96].

Optimizing garbage collectors

Garbage collection algorithms may be evaluated on three different axes: application throughput, memory space overhead, and pause time. Application throughput may be measured in transactions per second (or equivalent) on a particular benchmark, or can be considered as the fraction of the machine resources (e.g. CPU time) dedicated to real application work rather than garbage collector or GC barrier overhead. This "real work" is called *mutation* in GC terminology, as it involves changing the heap by allocating and modifying objects, and it is carried out by a *mutator*; in contrast, the garbage collector operates under the covers as far as the application is concerned. Space overhead determines the cost of a machine (in DRAM parts and power), and has two aspects: the amount of garbage which is allowed to accumulate in memory before the garbage collector is invoked (or before it can return that memory to be recycled), and the amount of garbage which could have been reclaimed but was not (so-called *floating garbage*) because of conservative assumptions in the collector. The former is partly a scheduling issue, the latter depends on GC algorithm design and the behavior of a particular application. Pause time requirements are critical whenever an application interacts with the real world, from financial trading rooms (where seconds are worth millions of dollars) to simple web applications (a server which takes longer than 5 or 10 seconds to respond is often abandoned as unusable by many customers). Pause time requirements may be specified either as a (soft) real-time limit ("no GC pauses longer than 10ms"), or a somewhat flexible service level ("90% of pauses less than 100ms") - and in some batch-type applications, there may be no particular pause time requirement. Hard real-time applications, where proof of response time is required, are an entirely different kettle of fish, and use custom-designed JVMs and compilation schemes with a different programming model for memory use (see the Real-Time Specification for Java). We won't consider hard real-time further here.

Most existing garbage collectors can optimize at best two of the three desirable attributes. If the only goal is application throughput, the most efficient garbage collectors will collect as infrequently as possible, pausing the whole application for the duration of the collection. In such a system the throughput can be traded off against space overhead across a wide range: a fairly simple argument will show that either time overhead or space overhead can be reduced to an arbitrarily low value, at potentially enormous cost in the other. A throughput-optimized-at-all-

Draft report - Please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

costs machine will therefore be enormously overprovisioned in memory (relative to the actual live memory requirements of the application), and few customers are willing to buy such a machine. Pauses in such a system may be terrible, but a 30-minute GC pause which occurs once a day is only a 2% penalty for throughput.

More complex garbage collectors try to slice the GC salami more thinly, to reduce GC pauses. This means spreading the GC work more evenly throughout the mutator execution, even if the result is lower throughput. There are three general techniques which may be combined. *Incremental* collectors suspend mutation fairly frequently to perform a small piece of the GC work. *Generational* collectors partition the heap into regions of newly-allocated and old objects, performing collections more frequently on the new objects based on the hypothesis (experimentally good) that most objects are likely to die young. Thus, generational collectors not only perform GC work in smaller chunks (as new space is much quicker to collect than old space) but actually reduce the amount of GC work in total (as effort is concentrated in the region where most garbage is likely to be reclaimed). Finally, *concurrent* collectors perform garbage collection in a separate thread while mutation occurs. The goal of concurrent collectors is to avoid lengthy *stop-the-world* pauses, in which all the mutators are suspended; it is probably unavoidable that each mutator thread must be stopped individually at some point during the GC cycle.

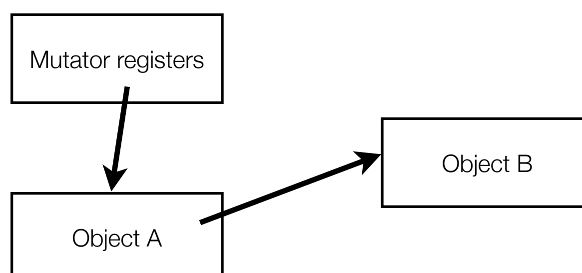
Incremental GC is somewhat outdated now, having been developed in the context of single-threaded systems like Lisp and Smalltalk. Java programs may contain many (hundreds or thousands) mutator threads, and bringing them all to a consistent state for garbage collection to take place (a *safepoint*) is a significant undertaking. Thus, suspending mutation completely for GC has a high overhead; doing that frequently, to perform just a small amount of work, is not efficient. The modern goal is fully concurrent GC, but the issue here is how to ensure correctness: we must ensure that the collector cannot get out of sync with the mutators, an exercise in concurrent programming. Some collectors use *mostly-concurrent* collection [Boehm91, Printezis00], where the collector performs most of its work concurrently but there is still a stop-the-world pause at the end to ensure correctness; the duration of the stop-the-world phase depends on how well the concurrent phase anticipated what the correct GC would do. Lock-based synchronization between the mutators and collector is not really appropriate (the limiting case of a “whole heap lock” is just non-concurrent GC!), so concurrent GC techniques rely on fine-grained communication between the mutators and collectors.

An example of concurrency failure

An example will illustrate the kind of situation that the GC designer must prevent.

A garbage collector needs some way to determine the liveness (or otherwise) of objects. An object is live if it is potentially reachable by a mutator thread. At any given moment a mutator thread will hold some references in its registers and stack, and can obtain other references from some known globally-visible locations (e.g. static variables) - these references “in hand” are called *roots*. Some of the root-referenced objects will contain references to other objects; the reachable objects are the transitive closure in the object graph following references from the

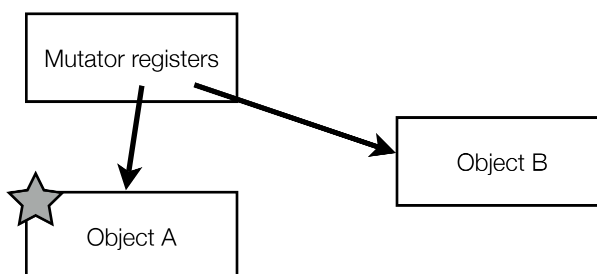
roots. A common GC technique (sometimes called 'tracing') is to collect roots from the mutators by looking at their registers and stacks, then recurse over the heap following references from one object to the next and marking the ones that have already been seen. When the recursive marking is complete, every unmarked object is garbage and can be collected (this is 'mark-sweep GC'). A naive attempt at this in a concurrent system can fail, as in this example:



In the beginning, the mutator holds a reference to object A in a register (let's say %i0), and object B is referenced from some field (say at offset 40) in object A. The GC collects the roots by suspending the mutator thread at a safepoint and examining its registers, so now it knows that object A is alive and marks it. Let's suppose that the mutator restarts and executes the following code:

```
ldx [%i0, 40], %i1
stx %g0, [%i0, 40]
```

That is, the mutator reads the reference to object B out of object A, then overwrites that field. The object graph now looks like this:



Now the collector goes to look at object A and sees that there are no references in it. At this point the traversal is finished, and the GC believes that object B is dead (because it never saw a reference to it during the traversal). However, object B is actually still reachable by the mutator, a serious bug.

GC barriers

The general solution to the concurrency problem here is a *GC barrier* (which is unrelated to memory barriers like the MEMBAR instruction). A GC *write barrier* is executed by the mutator when it stores a reference into the heap, changing the object graph in such a way as to confuse the collector. A *read barrier* is executed when the mutator reads a reference from the heap into a register. The design of the barriers is central to any concurrent GC algorithm, and because they

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

are executed frequently during mutation (potentially on every load or store to the heap) their performance is significant. In the example above, the particular concurrency problem could have been solved with either a read or a write barrier. A read barrier associated with the `ldx` could inform the collector that the mutator saw a reference to object B; alternatively, a write barrier with the `stx` could report the old value (i.e., the reference to B) that was overwritten. Either would ensure that the collector does not miss seeing the reference to B. "Informing the collector" in this case could mean storing the appropriate value into a buffer (a *logging* barrier), but other kinds of barriers may mark the referenced or referring object.

Software implementations of barriers may be expensive, both in time to execute and in the size of generated code (these are instructions which will be emitted in the native code by the JVM's compiler/code generator). In software, the pressure is therefore to make simple barriers, but existing proposals for software-only barriers for concurrent GC are quite large and complex [e.g. Bacon01, Hudson01]. However, in practice only a small fraction of the barrier executions are actually significant for correctness (remember, we only need to catch the reads or writes which *might* confuse the collector). Much of the time the collector may not be running, or the mutator may be reading a reference which the collector already knows about. GC barriers may thus be conditional, to reduce the number of times that the barrier proper is actually activated, but the overhead of testing the condition in software may itself be significant: it occurs on every load or store of a reference.

Termination

We considered above how read and/or write barriers can ensure that the collector does not get confused by the mutator into losing track of live objects. For example, a read barrier which reports to the collector every reference read by a mutator would solve the problem. However, such an algorithm would be completely impractical: as the mutators execute they continue generating an endless stream of references (most of which the collector already knows about), so that the collector never reaches a stable point at which it can determine conclusively what is garbage. When the mutators can continuously generate work for the collector we have a termination problem.

A simple, conventional solution to the termination problem is to introduce a stop-the-world pause (for example, mostly-concurrent GC). With all the mutators paused, the collector can catch up with its processing without the mutators generating more work. However, this is completely against the goals of fully-concurrent GC. One reason is that the length of the stop-the-world pause is potentially enormous: consider that, for example, the final reference exposed to the GC by a mutator before the pause is in fact the root of an otherwise-unreachable linked list through the whole of memory.

Conditional GC barriers assist greatly with the termination problem. If the barrier (executed in the mutator) can determine whether the GC already knows about a reference then we can bound the amount of work which the mutators will give to the collector (because each reference will be examined at most once).

Relocation

The ability to relocate objects is very important in an object-oriented system, because otherwise the memory space becomes fragmented over time. Fragmentation occurs when the allocation of new objects can only occur in the holes left behind by old garbage objects; in general the holes will get smaller and smaller, as a few long-lived objects break up the space. Fragmentation is avoided by *compaction*, in which live objects are relocated into a contiguous region, leaving a large empty area for future object allocation.

Many GC algorithms also incorporate a compaction phase, although the two operations are logically distinct. It's often convenient to perform compaction as part of GC, because in conventional systems one must locate all of the references to an object in order to relocate it (because a reference to an object is its virtual memory address), and the garbage collector also works by finding references to objects. Notice the important distinction from GC though: to determine liveness it's sufficient to find one reference to an object, whereas relocation requires updating all references. Concurrent relocation with a conventional addressing scheme is a difficult problem, and almost impossible without special hardware support: it is hard to "atomically" move an object to a new location and update all the references to it.

Hardware barriers

As we have seen, software barriers for concurrent GC suffer the problems that (i) conditional barriers may contain a lot of code and/or be expensive to execute, and (ii) unconditional barriers may prevent reliable termination of the algorithm. Conditional operations are much easier to perform in hardware, if the information on which to base the decision is ready at hand. The barrier is expected to be activated infrequently, so a relatively expensive but uncommon path may be appropriate (e.g. a trap to a software handler). On the other hand, the common case of testing the barrier but performing no action can occur very efficiently. The trick to obtaining efficient mutation with a concurrent GC is therefore exposing appropriate state to allow hardware implementation of the barriers.

Summary

Concurrent garbage collection and compaction are necessary to get acceptable pause times with very large heaps and multithreaded applications. Conventional concurrent programming techniques (locks, etc.) aren't well suited to GC barriers because they add mutator overhead in the common case.

2. The Maxwell approach

The Maxwell project has been investigating hardware support for concurrent GC and relocation, which may solve many of the problems described above. This document describes many GC techniques, applied to a quite ordinary but time-honored underlying heap structure: generational collection, with a semi-space copying new-space and a mark-sweep old-space. The goal throughout, however, has been to push scalability to extremes, by eliminating completely all stop-the-world pauses, decoupling new-space and old-space GC cycles, separating compaction from collection (in old space), and offloading much of the old-space collection to GC units down in the memory system. A design principle is that mutator threads approaching a synchronization

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

point (e.g. advancing towards a safepoint at which they will report their register and stack contents to the collector) will not be impeded by those which have already passed that point; this ensures forward progress and termination. A secondary principle (sometimes in conflict) is that, as far as possible, the GC pauses experienced by a mutator thread will be independent of other threads' actions (ignoring scheduling issues).

It is quite possible that other system organizations are possible using similar techniques. For example, the semi-space copying algorithm presented here may adapt quite readily to a Garbage First-style heap [Detlefs04]. There are too many implementation details to consider every variation. What follows is the grandest Maxwell design, to illustrate all of the techniques applied simultaneously. Experience under simulation, targeted at a particular system and workload, may well show that subsets of the techniques perform better and are easier to implement.

3. Object-addressed memory

Maxwell's object addressing scheme serves two purposes: it exposes old-space objects and their contents to the hardware, which allows offloading to hardware of some of the GC work; and it enables concurrent relocation, because objects are no longer referenced by their memory addresses. The object addressing scheme is based on that of the earlier Mayhem project [Wright06].

There are two kinds of objects in the Maxwell system. A reference to the first kind is its virtual memory address (as in conventional systems), but the second kind of object is referred to by a location-independent *Object ID* (OID). The remainder of this section discusses OID-referenced objects; we'll return to the other objects later.

The actual location (in physical memory) of an OID-referenced object is stored in one place in the system, the *object table*, which is indexed by the OID. The object table is somewhat analogous to a page table in a paged virtual memory system; an object's object table entry gives its physical address in memory, object size information, and some other metadata.

Load or store instructions directed to object fields present two address operands, the object ID and the field offset. Unlike normal SPARC® virtual addresses, the two operands are not added. Instead, they're logically concatenated to make an (object ID, offset) pair, which we call an *encoded address*; the actual encoding is described later. This address bypasses the TLB (OID-referenced objects are not part of the paged virtual memory space), and directly indexes the caches. Encoded object addresses are distinguishable from ordinary physical memory addresses; the encoded addresses are longer than conventional physical addresses, so we can effectively map physical addresses into a small portion of the encoded address space which is not used by objects (e.g. all zero high-order bits). The caches become "encoded indexed, encoded tagged" (analogous to "virtually indexed, virtually tagged" conventional caches in the case of object access). There is thus no need to translate an object address into a physical memory address in the case of a cache hit, and cache coherence operations work normally on the encoded

addresses[1]. Object addresses do not alias (each object is accessible through only one OID), so a common problem with virtually-addressed caches does not apply. With the physical address space contained in the encoded address space, the cache can contain copies of any mixture of objects and ordinary physical memory.

Most objects are small, with only a few fields, but the Java language does permit very large objects, up to 64k fields or 2B array elements. To avoid carrying around so many offset bits when very few objects will use them, we limit hardware-supported objects to a certain size (say 1kB, or 10 offset bits). Larger objects are handled differently (see later).

Translation

Encoded addresses are all very well for indexing and tagging the caches, but after a cache miss we must still find the object's contents in memory. A piece of hardware, the *translator*, resides after the last level in the cache hierarchy, and maps cache line fetch and writeback operations on object addresses to their equivalents in physical addresses. The translator is the primary user of the object table: when a request comes in to fetch an object cache line identified by an encoded (OID, offset) pair, the translator inverts the encoding to reveal the OID, then looks up the physical memory address of that object in the object table. In the case of a cache miss, the translator then constructs the requested (OID, offset)-tagged cache line by reading from the physical memory copy of the object; cache evictions work in the opposite direction.

The physical memory backing the object is in normal operation only touched by the translator, and is not kept coherent with any object-addressed cache lines which may exist. The mutator threads only operate on the object-addressed version.

We don't insist that objects perfectly align in memory on physical cache line boundaries – most objects are small, and the waste in DRAM would be intolerable. Instead, an object cache line may overlap two physical cache lines. Thus, fetching an object cache line may cause one or two fetches of cache-line-sized chunks from the DRAM (partial cache line fetches may be an option). The object table entry also contains object size information, so the translator can determine where the last cache line in an object ends and avoid modifying physical memory past the end of the object. The size information provides slightly imprecise bounds checking on objects: a write just past the end of an object (within the final cache line) may appear to succeed in the cache, but will not be written back to memory. A cache line fill or eviction which is completely off the end of the object may be rejected by the translator as an error.

Encoding

The encoding function, which packs an OID and an offset into the address which will index and tag the caches, must be very implementation-specific. Cache indices and perhaps banks are commonly calculated by bit extraction from the address, and encoded object addresses have peculiar properties because the (OID, offset) space is sparsely populated. In particular, we would like the multiple cache lines of a large object to be spread throughout the cache, so that they do

---

[1] This is true in snooping systems, but directory protocols relying on a home node need some extra work which we don't discuss further here.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

not all conflict within a single cache set. However, we would also like a working set of small (single cache line) objects to be able to fill the cache. In principle some of these deficiencies could be corrected by calculating more complex cache index functions for each cache, but the conventional method of simple bit extraction to form the index is very attractive. In neither case does simple permutation of the OID and offset bits produce the desired properties.

Our proposed design for an encoding function is very simple. The low-order OID bits are XOR'd into the high-order offset bits (after the byte-within-cache-line bits), and also duplicated at the high end of the encoded address. That is, using a 40-bit OID and 10-bit offset with 64-byte cache lines as an example:
encoded_address := { OID[3:0], OID[39:4], (OID[3:0] ^ offset[9:6]), offset[5:0] }

One could also imagine using more complex invertible functions like addition.

With a more complex memory addressing structure, e.g. striping addresses across banks and multiple memory controllers, one may wish all the cache lines of the same object to go to the same memory controller/translator. In that case the scheme must be designed with the particular striping scheme in mind, to ensure that only OID bits are used to map to the bank.

Concurrent relocation

The basic scheme for relocation is the same however it is carried out: copy the entire contents of the object from the current physical address to a new physical address, and update the object table entry to point to the new location. However, this must happen atomically with respect to mutator operations: that is, mutator reads must not see a partially-copied object, and mutator writes must be reflected in the new copy.

One can imagine the copy operation itself being performed either in hardware or in software. If the translator is responsible for copying in hardware, it may synchronize the copy operation with cache line fetches or evictions caused by the mutator, for example by delaying the mutator operations until the copy is finished, or (with more predictable latency) by using its knowledge of how far the copy has progressed to direct operations to the new or old location as appropriate. A software relocation is also possible, as long as synchronization information is kept in the object table entry. It's enough for the object table entry to contain a "modified" bit, which (as with page table entries) is set by the translator when updating the physical memory backing the object. A software copy can proceed by clearing the modified bit, copying the data, and then CAS'ing in the new location to the object table entry. The CAS will fail if the modified bit was meanwhile set by the translator, indicating that the copy must be reattempted. Eventual success of the copy can be guaranteed by suspending whichever mutator thread is interfering.

Finding references

The system described so far stores objects in memory and uses object-addressed caches, and allows concurrent relocation. If we are to offload GC work to the hardware, one must be able to locate the references in objects. Conventional systems maintain software data structures associated with the object's class, but this is not convenient for offload processing – we need information which is "to hand" in the hardware. There are two solutions known in the literature:

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

tagged memory, in which each memory word has a "65th bit" to distinguish references from non-references, and a bidirectional object layout [Wolczko99, Gagnon01]. Conventional object layouts place the object header at the beginning of the object (offset zero), with fields following afterwards; each subclass inherits the layout of its parent and tacks its own newly-declared fields onto the end. In a bidirectional layout the fields extend in both directions from the header, with references at negative offsets and non-references at positive offsets; the object table entry contains size information in both directions. It is thus easy to locate the references within an object without having to understand any class data structure.

4. Object allocation, virtually-addressed objects, and "new space"

Allocating OID-referenced objects is a moderately expensive operation: a free OID must be found (perhaps from a 'free list'), an appropriately-sized chunk of physical memory allocated, and the object table entry is then populated with the object's size and location. Many Java objects are created and quickly die (recall the 'generational hypothesis'), so their allocation and reclamation is a high proportion of their total lifetime. Further, their short lifetime means they will probably not take advantage of the concurrent relocation offered by the indirect object addressing.

A more efficient allocation mechanism is *pointer bumping*, universally used in production JVMs. The allocation pointer points into an empty region of virtual memory (an *allocation buffer*, which in some refinements is a *thread*- or *processor-local allocation buffer*, TLAB or PLAB); to allocate an object of size N bytes we just take the current allocation pointer as the object reference, then bump the allocation pointer by N. This allocation style complements *scavenging* GC extremely well: a scavenging collector relocates live objects out of an area, leaving behind a region containing only garbage, which is ideal for pointer-bumping allocation. Scavenging also has the advantage that it only touches the live objects, which in new space tend to be greatly outnumbered by the dead objects.

The combination of pointer-bumping allocation and scavenging collection is so desirable that in Maxwell we allow some objects to be referenced by their virtual addresses (just as in a conventional JVM) – this portion of the heap is called *VA space*, a subset of which is *new space* where recently-allocated objects reside. An object is not given an OID until it has survived some number of new-space garbage collections, in which case it is *tenured* into *old-space*, by allocating an OID and physical memory for it as described above. Large objects, those too big for the number of offset bits, also live in VA space.

In Maxwell, therefore, an object reference may be either a virtual address or an OID. Load and store instructions distinguish virtual addresses from OIDs by the high bit(s) (e.g. bit 63) of the first source register (there are alternatives); this costs half of the virtual address space in the JVM process.

New space GC

The new-space garbage collector uses a concurrent semi-space copying algorithm. The basic semi-space algorithm (a concurrent version of Cheney's algorithm, see section 6.1 of [Jones96])

proceeds as follows: Initially, all new space objects are in the ‘from’ semi-space; the ‘to’ semi-space is empty. As the garbage collector discovers that objects are live, it copies them blindly from from-space into to-space, leaving behind a *forwarding pointer* in the old location indicating to where the object was moved; note that the to-space copy may still contain pointers back into from-space. A second wave of processing follows along inside to-space examining the fields in the copied objects: whenever it encounters a pointer into from-space it either indicates a newly encountered object to copy, or an object which has already been copied, but in either case the pointer into from-space is replaced by the equivalent pointer into to-space. Eventually there are no more live objects to copy, and all the pointers contained in to-space objects also point into to-space. At that point from-space contains only garbage and can be recycled. Roots for new-space GC are mutator registers and stack locations, plus any references in old-space which point to new-space objects (the old→new references are tracked by an inter-generational write barrier). All the roots of course must also be updated to point to the new to-space copies.

The new-space collector moves objects and updates the references to them, so as well as concurrent GC we also have a concurrent relocation problem: we must ensure that a mutator does not use the old from-space version of the object when it has been moved to to-space.

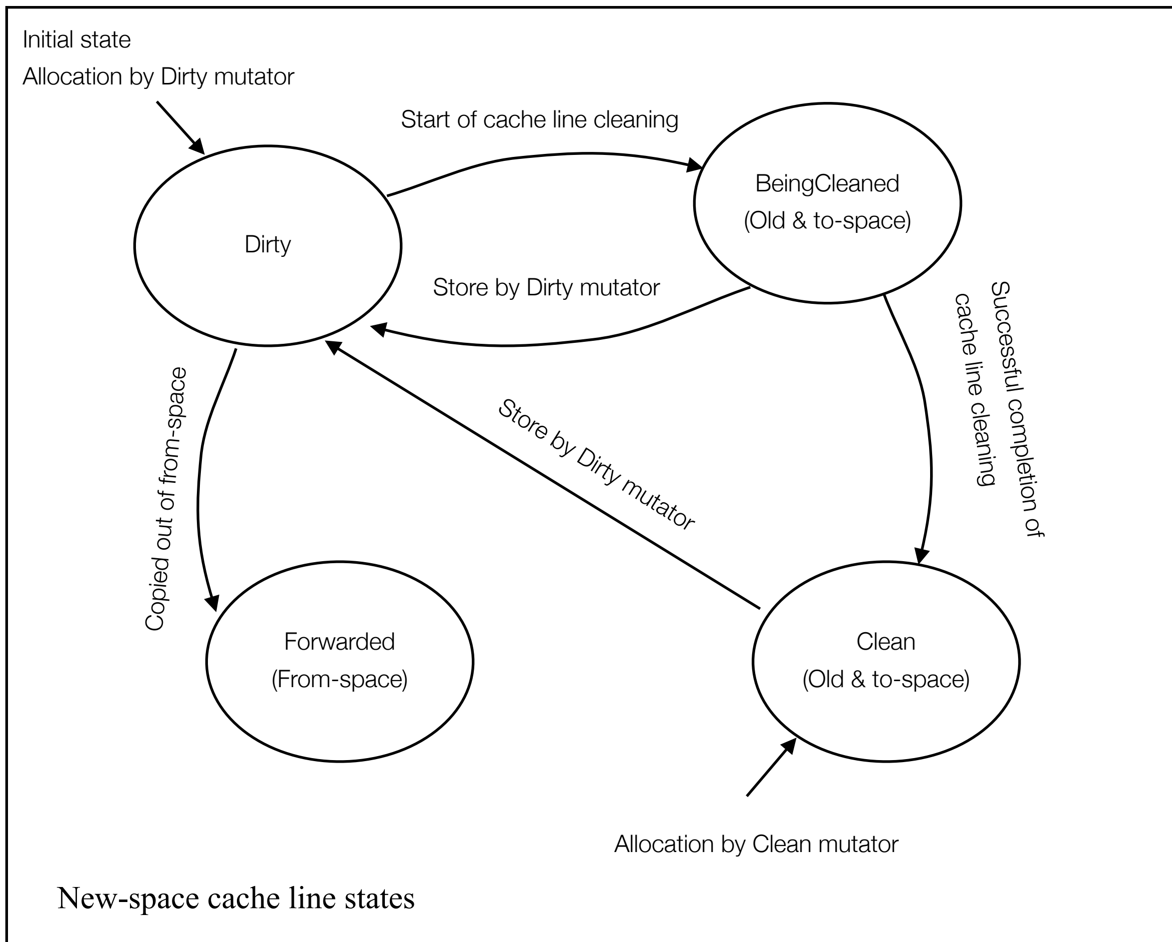
New-space GC state

Concurrency is controlled by hardware GC barriers built into the mutator’s load and store instructions. The barriers compare the state of the mutator thread with some metadata stored alongside each cache line (e.g. with the ECC tags); the GC bits are part of the coherent state of the cache line, and may be modified by certain write barriers (but not read barriers), so the usual coherence mechanisms apply just as they do to data.

A mutator thread, for new-space GC purposes, is in one of two states: Dirty or Clean. A Dirty mutator thread may hold or use references to either from-space or to-space objects. A Clean mutator thread may only have references into to-space (so the actions of a Clean mutator cannot interfere with the termination of the garbage collector).

A cache line exists in one of four states: Dirty, Clean, BeingCleaned or Forwarded. A Dirty cache line may contain pointers into from- or to-space. A Clean cache line contains only pointers into to-space. The BeingCleaned intermediate state exists so that cache lines can be cleaned concurrently; it acts as Clean with respect to stores and Dirty with respect to loads. Finally, the Forwarded state prevents mutators from touching the cache line at all; it indicates part of an object in from-space which has already been moved to to-space. In principle the four states apply to all cache lines in the heap; however, only from-space lines will ever be genuinely Forwarded. Java stacks are private to their threads, so (like registers) they do not need GC concurrency protection. The 4 cache line states can be encoded in two bits per cache line, although the interpretation of the bits is different in even and odd GC cycles (see later). There is one bit per thread (or core/processor if desired) to indicate the GC phase; if the phase bit is not per-thread, then it should also invert the interpretation of the thread’s Dirty/Clean bit.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only



New-space cache line states

New space barrier actions

The following barrier actions apply only to reads and writes of references; non-reference loads and stores do not care about the Clean/Dirty state of the thread or cache line, but will still trap if a mutator attempts an operation on a Forwarded cache line. New instructions (`LDR/STR`) distinguish reference loads or stores from the usual 64-bit SPARC load/store instructions `LDX/STX`.

Thread state	Cache line state		
	Dirty / BeingCleaned	Forwarded	Clean
Dirty	Reads: OK Writes: set cache line to Dirty	Trap	Reads: OK Writes: set cache line to Dirty
Clean	Reads: Trap Writes: OK	Error	Reads: OK Writes: OK
(GC)	Reads & writes OK		

The new-space barrier operations are very simple: writes by a Dirty mutator may set a cache line to Dirty, plus there are two cases where a trap must be delivered. Firstly, a thread attempting to touch a Forwarded cache line must be prevented from doing so: this is the out-of-date from-space version of an object which has been moved to to-space. The trap handler will find the forwarding pointer in from-space and update the register containing the from-space reference before re-trying the offending load or store. Secondly, a Clean thread is not allowed to read from a Dirty or BeingCleaned cache line: that might lead to its seeing a reference which is unknown to the GC. The trap handler here will either clean the cache line itself (helping the collector along), or just wait for the collector to get to it eventually (which could take some time). When the cache line is clean, the operation may proceed. The cleaning operation is described below.

The remembered set

The new-space collector, like any generational system, must be able to find old→new references easily. The *remembered set* is maintained by a logging store barrier: whenever a store-reference instruction is executed with a destination address which is an OID and a datum which is a virtual address, the OID is logged somewhere accessible to the thread (and ultimately the collector). That could be directly to a memory buffer, or a small number of registers inside the CPU which are flushed to memory by software when they fill.

GC: Quiescent/Initial state

An the beginning of a GC cycle, every mutator thread is Dirty, all of from-space is Dirty, to-space is empty (we do not care about its state bits), cache lines in old space containing pointers into from-space (i.e. roots) are Dirty, and the rest of the heap (old space) may be either Dirty or Clean.

This initial state is almost completely stable: the read and write barriers do not disturb anything, with the minor exception of stores into old-space which may silently set the cache line to Dirty. We expect the majority of processing to occur in this state.

GC: Thread cleaning & root processing

A garbage collection is actually initiated when a mutator is suspended at a safepoint and cleaned: that is, all from-space objects referenced from its registers and stack are forwarded to to-space. Once that's done the mutator's state is switched to Clean, and mutation can resume. From now on this mutator will never see a reference into from-space. The remembered set log associated with this thread may also be handed off to the collector at this point.

As soon as one mutator has been cleaned, other mutators may start to see Forwarded cache lines.

There is no need to suspend all of the mutators at the same time (no stop-the-world pause), although every mutator must be cleaned once before the GC cycle can end.

The other source of roots (apart from the mutator threads) is old→new references. At some point during the GC cycle the collector iterates over this remembered set and cleans the old-space cache lines containing them. Cache line cleaning is described below.

Copying objects

The collector (or sometimes a mutator acting on its behalf) copies object from from-space to to-space as follows:

- Allocate sufficient space for the object in to-space
- Iterate over the cache lines in from-space containing the object:
    - Set the cache line to Forwarded
    - Copy each field in that cache line to to-space (as Dirty)
- Store the forwarding pointer (the new to-space address) into the header of the from-space version. (If there may be a race between multiple collector or mutator threads, this store should be a CAS to ensure that only one actually makes the definitive to-space copy.)

Note that the Forwarded state (as marked in the cache lines) may run off the start or end of the object into adjacent objects: this is allowable, merely conservative.

Any mutator attempting to touch a field in the from-space version will trap upon seeing the Forwarded state. The trap handler will forward the reference as follows:

- Check the object header to see whether a forwarding pointer has been installed
- If not: forward the object as described above (helping the collector along)
- Fix up the offending register (identified by looking at the code for the exception PC) with the new to-space address
- Retry the operation

The collector's cleaning pass

As soon as some objects are copied to to-space, the collector is free to start transitive forwarding by cleaning to-space cache lines. It iterates over the to-space cache lines from the beginning, identifying the object fields containing references (either via the object's class, or by the sign of the offset for bidirectional objects). Cleaning a cache line means ensuring that it contains no pointers into from-space. The cleaning operation proceeds as follows:

- Set the cache line to BeingCleaned
- For each reference field in the cache line:
    - If the reference is a pointer into from-space: forward the reference (as with the mutator register case above)
    - CAS the forwarded to-space reference into the field. There is no need to retry if the CAS fails.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

- If the cache line is still BeingCleaned, set the cache line to Clean (this is an atomic-type operation). Otherwise, the cleaning operation failed (because a Dirty mutator stored into the cache line). Whether to retry depends on why the cache line was being cleaned.

As long as some Dirty mutators exist, Clean (or BeingCleaned) cache lines may become dirty again because a Dirty mutator may store a reference into the cache line. The collector must therefore repeat the cleaning pass after all the mutators are cleaned, or indeed may choose to delay starting the cleaning pass until then. When there are no more Dirty mutators each cleaning operation will complete successfully.

The cleaning pass may be parallelized easily over multiple collector threads, because the operation on each cache line is independent. The only issue is finding the references; a conventional semi-space layout is parseable from the beginning because objects are densely packed end-to-end, but the question is how to jump 'into the middle' and find references. If the semi-space was filled as multiple allocation buffers, then operation can proceed in parallel on each buffer, for example.

There is an interesting race if two threads try to clean the same cache line at once, while a Dirty thread also stores into the line (the state may go Dirty - BeingCleaned - Dirty - BeingCleaned - Clean, without in fact being completely clean). Thus, which thread is actually cleaning a particular cache line may require some synchronization as long as Dirty mutators exist. (An alternative way to avoid the race would be to have Dirty mutators trap and if necessary forward the individual datum when storing into a BeingCleaned cache line, which can remain BeingCleaned instead of becoming Dirty. This has the disadvantage of impeding those Dirty mutators which are heading towards a safepoint, thereby lengthening the period of overlap between Dirty and Clean mutators.)

Allocation during GC

Mutators are free to continue to allocate during the collection cycle. A Dirty mutator may allocate into either from-space (if there is room) or to-space; the objects it creates will of course be Dirty when it stores into them. A Clean mutator may allocate only in to-space. When allocation buffers in to-space are zero'd they should be set Clean before they are released to a mutator; the resulting state then ends up correct whether Dirty or Clean mutators allocate into it. A 'block store init' (or 'data cache block zero') instruction can set the cache line state to agree with the thread's state.

GC: Termination

When all of the mutators are Clean, and the collector's cleaning pass has reached the end of to-space, and old-space cache lines mentioned in the remembered set have been cleaned, the new-space GC is over. Every cache line containing a reference to new-space is now Clean, so there are no accessible references to from-space anywhere in the system.

At this point, it is possible for the mutators still to be taking occasional traps when reading from Dirty cache lines. These cache lines must be part of old space, but they cannot contain any from-space references because the remembered set was cleaned; the Clean/Dirty state of old-space

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

cache lines that are not mentioned in the remembered set is arbitrary, because we don't want to touch the whole of old-space in each new-space collection cycle. Thus, the mutators may be taking spurious traps but this does not affect termination. If performance suffers because of these traps, we may consider (at extra expense) introducing a 'None' state for old space cache lines, to indicate that there are no new-space references within.

From-space is now dead and its contents will not be used again (it contains a mixture of Forwarded and Dirty cache lines). To-space contains only Clean cache lines, and old space may contain a mixture of Clean or Dirty lines (with occasional transient BeingCleaned lines).

The phase flip

With the GC finished, we can flip phase to start again. Swapping the roles of to-space and from-space is easy. Inverting the mutators' phase bits causes an interesting re-interpretation of all the cache line and mutator thread state bits:

Cache line state bits	Odd phase	Even phase
00	Dirty	Clean
01	BeingCleaned	
10	Clean	Dirty
11	Forwarded	

As soon as a mutator 'sees' the inverted phase bit, it transmutes from being a Clean mutator operating on a mostly Clean memory into a Dirty mutator operating on a mostly Dirty memory. There is no need for the phase switch to occur on all mutators simultaneously: they do not interfere.

The state after all mutators' phase bits are flipped is very close to our initial quiescent state. The only remaining housekeeping is to ensure that all old space cache lines mentioned in the remembered set are Dirty. It's enough for each mutator thread to iterate over old→new references which it's created since the last GC (these are in the thread's remembered set log), setting them to Dirty; this does not need to be done at a safepoint.

Large objects

Large objects (mostly arrays) also reside in VA-space. They can be treated as a non-moving (or infrequently-moving) part of new space, so the copying algorithm may become mark-sweep in the large object area. Very large objects may never need to be relocated; one virtual memory page is equivalent to another as far as physical memory is concerned, so only in the case of address space fragmentation would we have to compact. Occasional compaction on the 'medium-sized objects' (too large for OID-space, but not many virtual memory pages) may be desirable, in which case the copying algorithm applies. The only objects of real performance significance to the collector are large arrays of references; the contents of scalar arrays never need to be examined during GC, and large non-array objects are vanishingly infrequent.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

An alternative is to treat large objects as part of old-space, making them roots for the new-space collection. The OID→new space write barrier will not log references stored into large objects, so the whole of the large object space (at least non-array objects and reference arrays) must be scanned for new space roots during the collection cycle (as with the logged old space roots).

There are potential page-based optimizations which may apply to the large object area (depending on page mapping costs). If a multi-page object is relocated away, it may be possible to demap the interior pages instead of marking the cache lines forwarded, thus recycling physical memory a little sooner. Only the object header (containing the forwarding pointer) must remain. If forwarding pointers can be held 'off to the side', it may be possible to demap the header too. If large objects are roots for the new-space GC, then root summarization based on write protecting pages is another option (to avoid re-scanning the whole of every large object in each GC cycle).

Reference equality

In simple non-concurrent systems, pointer equality is an easy test: two references point to the same object if and only if they are bit-identical. It is a problem for many concurrently-relocating collectors: if an object can be moved, and a mutator may simultaneously hold both the from-space and to-space references (as in the above Maxwell scheme), then two references may refer to the same object even if the pointers are different. (Sapphire [Hudson01] discusses the problem in detail; it is possible that Azul's published algorithm [Click05] in its pauseless form may also suffer from this, but the paper does not mention the issue.)

One solution is to ensure that there is a safepoint after the load (or construction) of a reference and before the comparison operation, assuming that all threads can be notified "simultaneously" to stop at the next safepoint to clean themselves. A Clean mutator cannot see both from-space and to-space references at once (only the to-space reference), so bit-equality comparison is safe. Testing one reference at a time to ask "is this a pointer into from-space?", or comparing the two references to see if they are in the same generation, may offer another solution; however, there are still potential races in this approach (for example, recall that at some point from- and to-space flip), and any correct solution will need care in the compiler/code generator. This is a topic for future work.

Note that the most common comparison ("Is this reference null/non-null?") is not confused by relocation.

New space summary

The new-space collector uses a concurrent semi-space copying algorithm, with hardware support for the GC barriers. State bits on cache lines indicate stale copies of objects which have been relocated away, and which cache lines may contain references as yet unknown to the collector. Each mutator thread must (in addition to any barrier-induced work) stop twice per GC cycle, once to clean its registers and stack, and once to ensure that cache lines containing old→new references are marked appropriately.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

5. OID-referenced objects and "old space"

The basic idea behind OID-referenced objects was covered in the introduction. Exposing objects and references all the way through the memory system allows much of the low-level GC work to be offloaded from the CPUs.

As a reminder, "old space" consists of small OID-referenced objects plus some large objects in the virtual address space. Most of the discussion here will be on the OID-referenced objects.

Hardware GC units

A *GC unit* is located somewhere in the memory hierarchy between the processor and the DRAMs, perhaps on the main processor die, a system board ASIC, or even an FB-DIMM style 'AMB'. The GC unit (GCU) incorporates the functions of the translator, and also performs some GC work during translation operations and autonomously. Much of the GC unit's work occurs directly on the DRAM copy of the object instead of the object cache lines. There is therefore scope for optimizing its operation, perhaps using sub-cache line size operations to the DIMMs to save power, and reducing the amount of communication through the system. The GCU mostly does not need consistency with the main processor – it is free to operate on the in-memory copies even if the processor caches contain modified lines, because GC is a conservative operation. As long as consistency is achieved by the end of the GC cycle then live objects will not be collected by mistake.

The GC unit's work consists mainly of:

- Handling translation requests, i.e. constructing (OID, offset)-tagged cache lines by reading from physical DRAM after a processor cache miss, or writing them back to physical memory after a processor cache eviction.
- GC barrier processing, which is integrated with the translation operations.
- Recursive marking during GC, following references from one OID-referenced object to another.

The GC units only handle garbage collection for the OID-referenced objects; the large object area is managed in software, exchanging references with the GC units.

OID allocation

Most objects in OID space will get there as a result of tenuring by the new-space collector after they have survived some number of collection cycles. Pre-tenuring, or allocation directly into old space by the mutator, is also allowed, but we won't consider the policy decisions further here.

Allocation into OID space is as follows:

- Locate a free object OID and corresponding entry in the object table. This is probably via a "free list" structure.
- Allocate physical memory of an appropriate size.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

- Fill in the object table entry with the object's physical address and size information.

The object is then available for use by the mutator; zeroing of fields is discussed elsewhere in this document. Allocation into the large object space is conventional.

Old-space GC

The old-space collector is based on a mark-sweep algorithm. Like the new-space collector it is completely concurrent, and indeed the two collectors can operate simultaneously, so that multiple new-space collections can take place while an old-space GC is in progress.

Old-space GC state

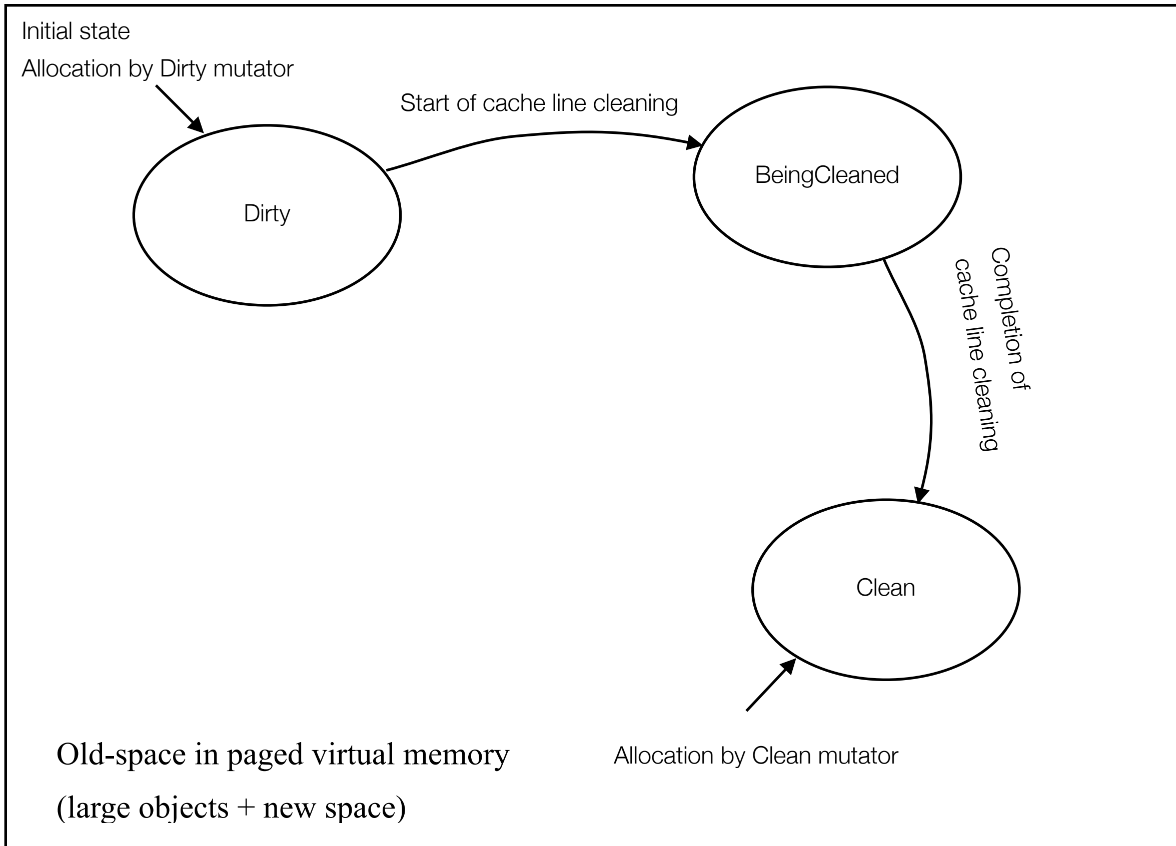
Like the new-space GC, the old-space GC requires two bits per cache line in the system (object or virtually-addressed) to hold state for the barriers. Each cache line can be in one of three states: Dirty, Clean, or BeingCleaned (the old space collector does not move objects, so there is no forwarded state). The BeingCleaned state applies only to the part of old space which lives in ordinary virtual memory (i.e. large objects), and is not used for OID-referenced objects.

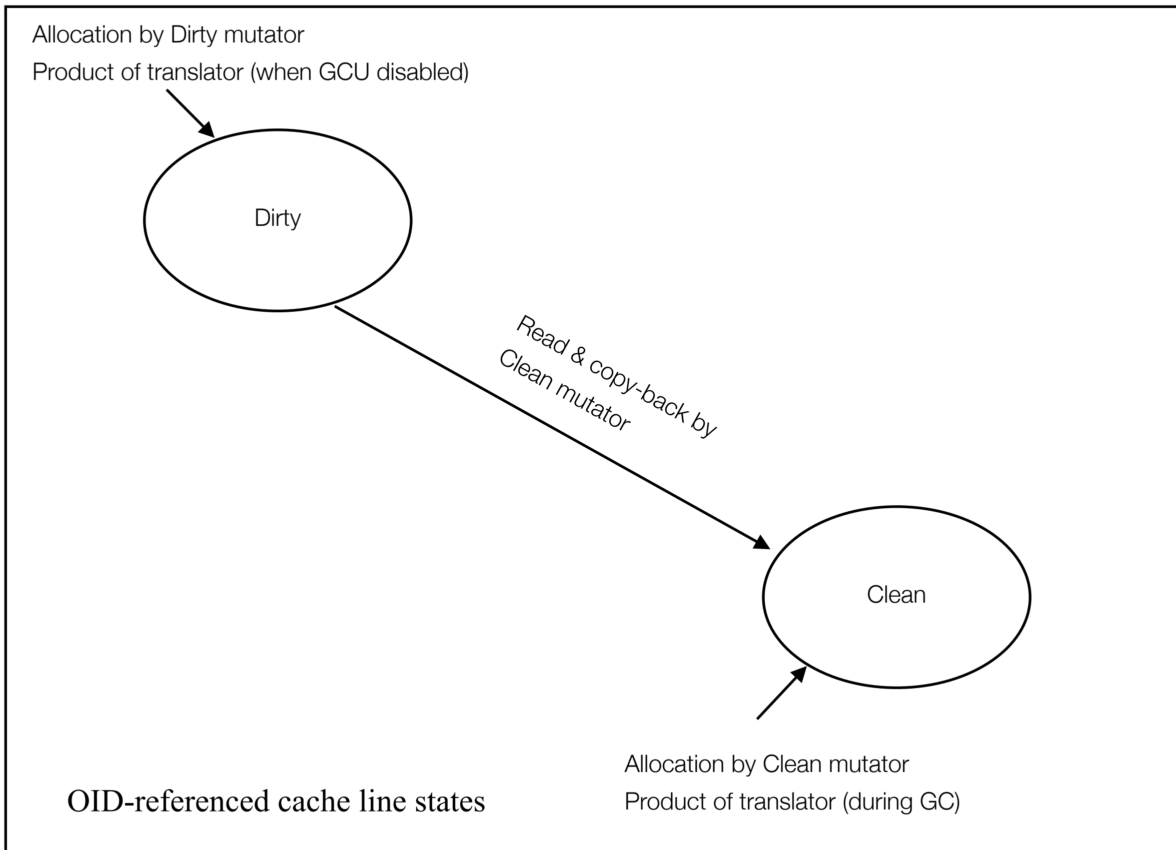
As with new-space GC, the state of an OID-referenced cache line is carried around by the coherence mechanism through the processors' memory hierarchy. However, there are two important differences. Firstly, the state of the cache line is also kept (along with a potentially stale copy of the data) in DRAM; the state bits of DRAM that holds OID-referenced objects contain the state of the object cache line overlapping the first word of the DRAM line. (The state bits for the first cache line of an object may be stored in the object table entry, if the first cache line starts part-way through a DRAM cache line). The GC units may read and update the stale state and data contained in DRAM, even if a modified version exists in the processor caches. Secondly, unlike the new-space state, the old-space state for OID-referenced objects may be inconsistent across the system, because in some cases a processor load may cause the GC state to be modified but will not cause a system coherence event.

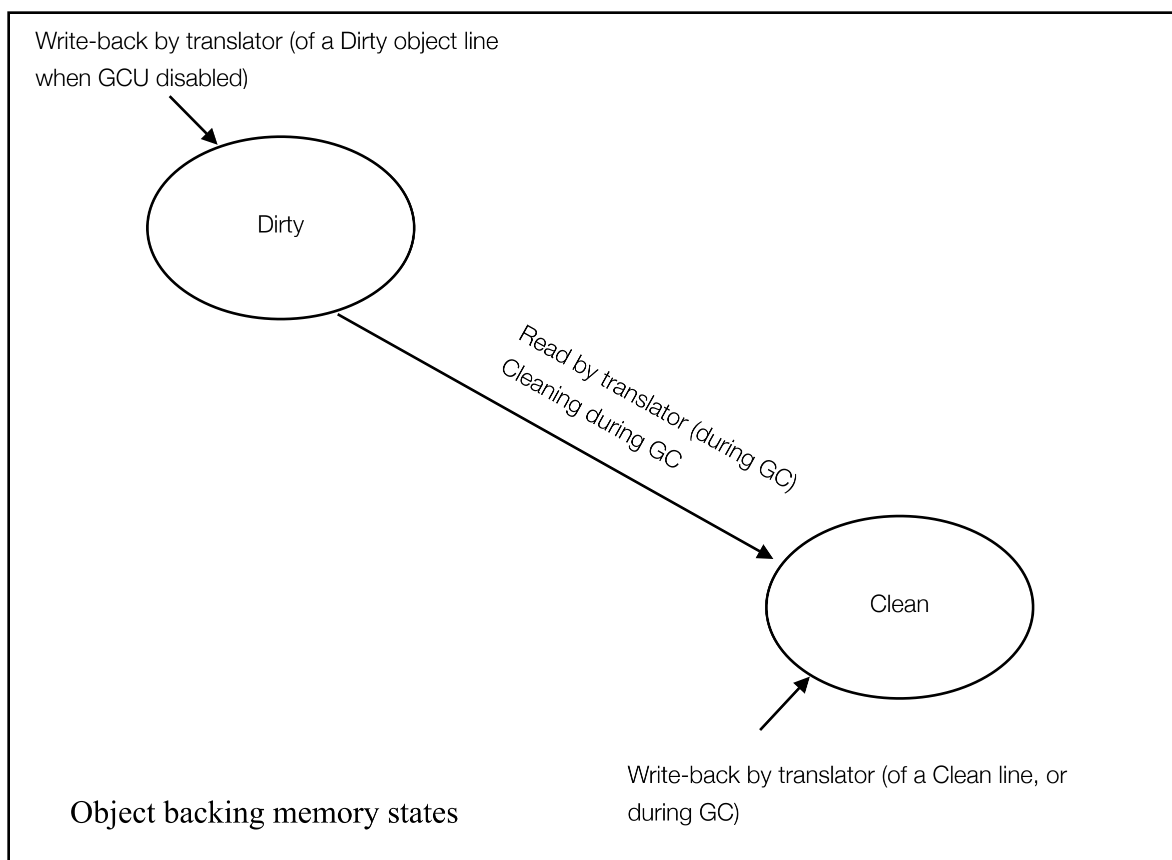
Each OID-referenced object has two bits in its object table entry, named Marked and Done. The Marked bit is set by the collector to indicate that the object is live. The Done bit is set when the GC unit has finished scanning the contents of the object for references to further objects.

Mutator threads are also Clean or Dirty for old-space GC purposes, and this state is orthogonal to the analogous new-space state. Just as with new space, the old-space collector interprets the state bits on cache lines differently in odd and even collection cycles.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only


Initial state
Allocation by Dirty mutator
Start of cache line cleaning
Dirty
BeingCleaned
Completion of cache line cleaning
Clean
Allocation by Clean mutator
Old-space in paged virtual memory
(large objects + new space)


Allocation by Dirty mutator
Product of translator (when GCU disabled)
Dirty
Read & copy-back by Clean mutator
Clean
Allocation by Clean mutator
Product of translator (during GC)
OID-referenced cache line states



Object backing memory states

Barriers

The majority of the old-space barrier processing is integrated with translation, so that it is invisible to the processors. Whenever the translator processes a read-request, it knows that anything referenced from the requested cache line is alive. Any references contained in the cache line and not already known to the GC (i.e., if the cache line is Dirty) will be sent for marking, and the state (in physical backing memory) changed to Clean. (This is similar to a snapshot-at-the-beginning algorithm (section 8.3 of [Jones96], and [Azatchi03]), operating on the physical backing memory). Object cache lines resulting from translation during GC are always Clean. Similarly, whenever the translator processes the writeback of a Dirty object cache line, any references it contains are sent for marking; this may be slightly conservative, based on the assumption that objects held in the processor caches must at least have been live recently. When the contents of the line are written to the physical backing memory they will be Clean. The action of the mutator, through the translator, always serves to advance GC, producing more Clean lines than there were before.

Some barrier processing still has to occur in the processors, to ensure termination. In this case, because there is no forwarding, loads and stores of non-reference values are not subject to the barrier.

<table>
<tr><th rowspan="2">Thread state</th><th colspan="2">Cache line state: OID-tagged cache lines</th></tr>
<tr><th>Dirty</th><th>Clean</th></tr>
<tr><td>Dirty</td><td>Reads: OK
Writes: OK</td><td>Reads: OK
Writes: Log</td></tr>
<tr><td>Clean</td><td>Reads: copy back cache line and set Clean
Writes: OK</td><td>Reads: OK
Writes: OK</td></tr>
<tr><td>(GC)</td><td colspan="2">Reads & writes OK</td></tr>
</table>

<table>
<tr><th rowspan="2">Thread state</th><th colspan="3">Cache line state: PA-tagged cache lines</th></tr>
<tr><th>Dirty</th><th>BeingCleaned</th><th>Clean</th></tr>
<tr><td>Dirty</td><td>Reads: OK
Writes: OK</td><td>Reads: OK
Writes: Log</td><td>Reads: OK
Writes: Log</td></tr>
<tr><td>Clean</td><td>Reads: Trap & clean cache line
Writes: OK</td><td>Reads: Trap & clean cache line
Writes: OK</td><td>Reads: OK
Writes: OK</td></tr>
<tr><td>(GC)</td><td colspan="3">Reads & writes OK</td></tr>
</table>

The “Trap and clean cache line” read barrier is exactly the same as the new-space case, except based on the old-space thread and cache line state, and references encountered must be sent to the GCUs or software old-space collector as appropriate.

The “Copy back cache line and set Clean” read barrier is an alternative to trap and clean, but only for OID-referenced lines: copying the line back to memory will cause the translator to clean the line. Modifying state bits on a read is unusual, which may cause some implementation problems. The software-equivalent trap and clean should still work, at the cost of more traps and with the usual care about atomicity of cleaning (BeingCleaned state, etc.).

The “Log” action is identical to the generational write barrier for new-space collection – the write proceeds but also gets recorded for GC purposes. References in a mutator thread’s log will be sent for marking when the thread is cleaned. The alternative to logging, as with new space, is to Dirty the cache line, but that has the disadvantage that Clean mutators may need to re-clean the line. Logging also removes the race when two mutators try to clean the same cache line at once.

Old-space GC: initial state

At the beginning of a GC cycle, all mutators are Dirty, and every reference-containing cache line in the heap (new space and old space) is Dirty, both in the processor caches and in memory. The GC units are disabled, so they do not clean cache lines during translation. Every object is not

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

Marked and not Done. This state is stable, and the processor's GC barriers will perform no actions.

Starting a cycle

An old-space GC cycle starts when the barriers in the GC units are enabled. From then on, translation will produce Clean object cache lines, and may trigger recursive marking. Dirty mutators which write into the Clean cache lines will also start logging their writes.

Mutator thread cleaning

At some point, each mutator thread must be cleaned. As with new space (and possibly integrated with a new space cleaning operation) all the references in registers and on the stack, plus any logged writes, are made known to the GC, then the thread is switched to Clean and allowed to continue. From this point on, the thread will never read a reference unknown to the collector. There is no stop-the-world pause, but each thread must be cleaned once during the GC cycle.

After all the threads are cleaned, any modified (actually M or O in MOESI) OID-referenced Dirty cache lines should be copied back from the processor caches to memory, so that the GC barrier gets to process the up-to-date contents. A cache flush is the unsubtle way to do this. (Some care may be needed with cache-to-cache transfers to ensure they don't miss the flush.)

Marking

The core idea of mark-sweep is very simple, and was described in the introduction. An object's Mark bit and Done bit are held in its OTE. Initially all objects' Mark and Done bits are clear. Marking starts from the roots (mutator thread registers and stacks, plus some permanent objects known to the VM); the mutator thread roots can be collected during a new-space GC. Other roots for old-space GC include any modified object cache lines held in the processor caches when the GC is initiated: many will be caught by the GC barrier in the translator (see later) when the cache lines are written back, but at some point (after all the mutators are cleaned) during the old-space GC cycle the processor caches must be flushed of modified object cache lines to ensure that the barrier sees them all.

When some roots have been collected they can be handed off to the GC units, which enter a marking loop:

- Remove a reference R from the set of objects to be marked
- Look up R's object table entry. If the object is already marked, we are done with this object.
- Otherwise, set the Mark bit in R's OTE.
- For each Dirty cache line in the object which may contain references:
    - Add each reference to the set of objects to be marked
- Set the Done bit in R's OTE.

The set of outstanding references waiting to be marked may be maintained in queues at the GCUs. A reference to be marked is sent from one GCU to another (possibly filtered by a local filter cache) across an inter-GCU network, where it is added to queue of incoming mark requests. The processing of these requests may be interleaved with locally-generated requests at the GCU. Flow control between the GCUs is TBD. In the event of queue overflow, it is permitted to mark an object in the object table as live, but not process its contents, and set a flag to say that overflow occurred. A later scan of the object table will find marked but not-Done objects. Queue sizing and overflow frequency is of course application dependent, TBD.

Large objects (in the virtual address space) follow a similar algorithm, except that they are handled by software GC threads running on the processors. Mark and Done bits for VA-referenced objects are kept in the object header, because there is no object table entry.

Eventually all the queues of outstanding objects to mark are empty

Allocation into old space during GC

Objects may be allocated into old space (either as OID-referenced or VA-referenced), as long as the ordinary GC barrier actions are followed when installing the contents. A Dirty thread may create Dirty or Clean cache lines, while a Clean thread should use initially Clean cache lines. (Details depend on whether zero-and-allocate is available or not). Newly allocated objects should be not-Marked and not-Done when created by a Dirty thread, but Marked and Done when created by a Clean thread; they therefore do not obstruct termination of the GC algorithm.

Interaction of new and old-space GC

There are many implementation options for how the new and old-space collectors interact, with details to be worked out for a particular design point.

Multiple new-space GC cycles can take place during the potentially long old-space GC cycle. OID→new references will continue to be logged by the inter-generational barrier, and old→new references will be updated by the new-space collector. The new-space collector may promote objects into old-space, subject to old-space barrier processing during copying.

There is some question about how to treat the liveness of new-space objects for old-space GC. One solution is to declare all of new-space live for old-space GC purposes, by making new-space into old-space roots. Under this scheme, any cycles that loop through old and new-space will not be collected until the whole cycle has been promoted to old-space. An alternative solution is for the old-space collector to treat new-space along with the large object space, by marking through it. In this case the old-space GC threads will have to take care when handling objects which are relocated (or perhaps even promoted) by the new-space collector, i.e. old-space GC threads should be subject to the new-space 'forwarded' barrier. A further complexity in this scheme is that may be possible for the new-space collector to reclaim an object after the old-space collector has marked it live. This can occur when the two algorithms are starting from different sets of roots, or encountering the constantly-mutating object graph at different times. Again, if old-space GC threads are subject to the new-space barriers this should not present a problem (careful programming may be an alternative).

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

Mark phase termination & sweeping

After all the mutator threads have been cleaned, and cache flushing has been performed, termination of the marking phase no longer depends on mutator actions. The collector threads must agree with the GC units when all marking has been completed (exact scheme TBD).

At this point, every live old-space object is Marked and Done, in its object table entry or header as appropriate. Every live object cache line in the system is also Clean; for VA-referenced lines this means the canonical copy (in cache or memory) plus any shared cached copies, whereas for OID-referenced lines it means any cached copies plus the backing memory. The translator's GC barriers will be encountering no more Dirty lines, and can be disabled. Similarly, the Clean mutators will encounter only Clean cache lines.

The GC units and software now begin a pass over the old-space objects, looking for dead (i.e. unmarked) objects. The GC units may perform this for OID-referenced objects with a single pass over the object table; the collector must also make a pass over large object space in software. Processing of dead objects is TBD: their OIDs must be returned to the free list, and their physical memory returned to the allocator.

Integrated with this sweep pass, the collector may prepare for the next collection cycle by resetting the Mark and Done bits. [TBD - does the phase flip work correctly on these too, avoiding the need to clear?]

The mutators prepare for the next GC cycle by switching phase: as with new-space, that causes all Clean mutators seeing Clean cache lines instead to become Dirty mutators operating on Dirty cache lines.

Reference-counting assistance for old-space GC

Description TBD.

6. Summary of hardware operation

Memory - state

Two bits per cache line for new space GC state. (“CacheLine.NGC”)

Two bits per cache line for old space GC state. (“CacheLine.OGC”)

Object table entry

Each object table entry contains:

Physical address of the object's backing memory

Size (one for unidirectional, two for bidirectional object layout)

Marked bit

Done bit

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

NGC and OGC states for the first cache line of the object (three bits total)

Processor - instruction set & barrier operations

One bit in PSTATE (or other privileged register) to enable Java extensions for the JVM process

Bits to indicate GC phase (odd/even) and strand Dirty/Clean state for new-space and old-space GC. ("Strand.NGC / OGC")

Some mechanism (PSTATE, ASI, opcode, ...) for GC operations to bypass the barriers below (TBD whether old and new space barriers should be disabled together or separately)

A way for load/store instructions to distinguish OID references from virtual addresses (e.g. by high order bit), for encoding & TLB bypass.

`LDB, LDH, LDW, LDX` (etc.): load scalar

Trap if CacheLine.NGC == Forwarded

`LDR`: load reference (new instruction)

Trap

if CacheLine.NGC == Forwarded

or CacheLine.NGC ∊ {Dirty, BeingCleaned } && Strand.NGC == Clean

or CacheLine PA-tagged && CacheLine.OGC ∊ {Dirty, BeingCleaned } && Strand.OGC == Clean

Copy back and set CacheLine.OGC := Clean

if CacheLine OID-tagged && CacheLine.OGC == Dirty && Strand.OGC == Clean

`STB, STH, STW, STX` (etc.): store scalar

Trap if CacheLine.NGC == Forwarded

`STR`: store reference (new instruction)

Trap if CacheLine.NGC == Forwarded

Set CacheLine.NGC := Dirty

if CacheLine.NGC ∊ { Clean, BeingCleaned }&& Strand.NGC == Dirty

Log if

Datum is VA && Destination is OID

or CacheLine.OGC ∈ { Clean, BeingCleaned } && Strand.OGC == Dirty

Translator operation

Read-translation:

1. Accept OID and offset from CPU or GCU
2. Look up OTE for object in the object table
3. Construct requested cache line from DRAMs using base address stored in OTE
4. If (Barrier enabled && CacheLine.OGC == Dirty) then
    1. Send references in cache line to unsorted mark queue
    2. Change in-memory and newly-constructed state to Clean
5. Send the cache line to requester

Write-translation:

1. Accept cache line from CPU
2. Look up OTE for object in the object table
3. If (Barrier enabled && CacheLine.OGC == Dirty) then
    1. Send references to unsorted mark queue
    2. Change state to Clean
4. Store cache line to DRAMs

GC unit operation

Mark state clearing:

1. For each object table entry: Set !marked and !done
2. Flip OID-GC phase
3. Mutators are now allowed to clean themselves

Object table scan (one pass performed after mutators and caches cleaned, but other passes may be done earlier; later passes may be needed in case of mark queue overflow):

1. For each object table entry:

2. if (marked && !done) then Send OID to local mark queue

GC loop:

1. While (local mark queue not empty):
2. Pop OID from local mark queue and MS-mark it

MS-mark:

1. Obtain OTE from object table (via translator)
2. If object is done: nothing further to do
3. If object is unmarked: mark object in OTE (via translator)
4. For each cache line in the object containing references: submit read request to translator (to cause cleaning, if necessary)
5. When all cache lines have been cleaned: mark object Done in OTE (via translator)
6. If cleaning causes mark queue overflow in Step 4, set Overflow flag and break out of the loop: undone objects will be revisited later.

Mark phase termination:

1. If (Overflow flag set) then
    1. Clear Overflow flag
    2. Rescan object table
2. Global termination when all queues empty :)

Sweep phase:

1. For each object table entry:
2. If (unmarked) then Kill object

Kill object:

1. Reclaim storage & OTE (somehow)

Allocation:

1. Find a free OTE (e.g. free list)
2. Set Marked and Done in OTE (caution: race between handing reference back to mutator and state clearing at start of GC cycle)
3. Allocate storage of given size, store PA into OTE

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

4. Cache lines initially are Clean and null (TBD whether GC unit or mutators take care of this)

7. Options, optimizations and extensions

Things not described in this version of the document:

Zero-and-allocate (“data cache block zero” or “block-store-init”) in the mutator

Reference counting assistance in old space

Multiple GCUs and address mappings between them

Virtualization (may be possible to feed OIDs through the TLB)

System software implications (strand state, OS/hypervisor management)

8. Summary of TBDs

GCU mark queue sizing and flow control

Exact termination algorithm (agreement between software GC threads & GC units)

OID free list management

Physical memory management by GCUs

Object contents zeroing (zero-and-allocate vs. GC unit clearing)

Old and new space barrier disabling in GC-threads (do we need new-space GC threads to be subject to old-space barriers, or are all GC threads the same?)

References

- [Azatchi03] H. Azatchi, Y. Levanoni, H. Paz, E. Petrank. *An On-the-Fly Mark and Sweep Garbage Collector Based on Sliding Views*. Proc. OOPSLA 2003.
- [Bacon01] David F. Bacon, Clement R. Attanasio, Han B. Lee, V.T. Rajan, Stephen Smith, *Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector*. Proc. PLDI 2001.
- [Boehm91] Hans-J. Boehm, Alan J. Demers, Scott Shenker, *Mostly Parallel Garbage Collection*. Proc. PLDI 1991.
- [Click05] Cliff Click, Gil Tene, Michael Wolf, *The pauseless GC algorithm*. Proc. 1st VEE, June 2005.
- [Detlefs04] David Detlefs, Christine Flood, Steve Heller, Tony Printezis, *Garbage-first garbage collection*. Proc. ISMM 2004.
- [Gagnon01] E. Gagnon, L. Hendren, *SableVM: A Research Framework for the Efficient Execution of Java Bytecode*, Proc. JVM ’01, 2001.

Draft report - please do not redistribute. Sun Confidential/Proprietary: Internal Use Only

- [Hudson01] Richard L. Hudson, J. Eliot B. Moss, *Sapphire: Copying GC without stopping the world*. Proc. ACM Java Grande - ISCOPE 2001.
- [Jones96] Richard Jones, Rafael Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, 1996.
- [Printezis00] Tony Printezis, David Detlefs, *A generational mostly-concurrent garbage collector*, Proc. ISMM 2000.
- [Wolczko99] M. Wolczko, D. Ungar, *Method and apparatus for optimizing exact garbage collection using a bifurcated data structure*, US Patent 5,900,001. Issued May 4, 1999.
- [Wright06] Greg Wright, Matthew L. Seidl, Mario Wolczko, *An object-aware memory architecture*. Science of Computer Programming 62(2) 145-163. October 2006.

Sun, Sun Microsystems, the Sun logo, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun Microsystems, le logo Sun, et Java sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.