UPMC Sorbonne Universités

Lip6

Big-GC: A new garbage collector for big data

Internship report, April 2014 - Sept 2014

Georgios Boumis
georgios.boumis@lip6.fr

Supervisor: Gaël Thomas
Referent: Julien Sopena

Abstract

The rise of cloud computing has enabled governments and companies to store huge data sets. Today, one of the major performance bottlenecks processing these data sets is the garbage collector of managed runtime environments such as the Java virtual machine. The garbage collector must now scan heaps of tens of gigabytes of memory, which causes intolerable pauses to the user. A recent study that was conducted jointly with the University of Neuchatel showed that database NoSQL Cassandra could introduce pauses of up to 6 minutes with a heap of a hundred gigabytes.

The aim of this internship is to propose and study a new garbage collection algorithm that can support large memory loads. The algorithm starts from the principle that the garbage collector can scan objects concurrently with the application if these objects are not accessed by the application at the same time. The aim of the internship will be to find an algorithm to separate the heap into two parts, one with the objects accessed by the application , the other with non-accessed objects.

Developments will be made in the HotSpot Java Virtual Machine (JVM) in C++. It is therefore requested the applicant to have an excellent level of programming and knowledge in memory collection.

Contents

Context

This internship takes place within the REGAL team[1] of the Laboratoire d'informatique Paris 6 (LIP6)[2] of Université Pierre et Marie Curie (UPMC)[3]. The aim of this internship is to propose and study a new garbage collection algorithm that can support large memory loads. The supervisor of this internship is Gaël Thomas and the referent is Julien Sopena. This internship finalizes the second year of the Distributed Systems and Applications (SAR) Master's degree.

1 Introduction

Garbage collector speeds up software development while increasing security and reliability. Garbage collectors have been used into modern popular languages such as Java. However, they may introduce pauses in the application's execution as well as overhead, that reduces efficiency.

The Java programming language is used in large server applications. Such large server and enterprise applications have enormous amount of live heap data in the 10s and 100s of gigabytes. Throughput is important for these kind of applications but they also may have (soft real-time) constraints that, if not met, are likely to annoy customers. An example of large real time system is the analysis of large exchange data. In these systems that treat huge data, any kind of pause or impact on performance can have many severe financial implications.

Managed runtime environments are supplied with huge amount of cheap computing resources and memory capacity. Enterprise applications demand high responsiveness and scalability. Garbage collection algorithms that pause application threads, are limiting the responsiveness and the scalability of the application. As live sets and heap sizes continue to grow and the time bounds continue to tighten, pauses become unacceptable.

Traditional "stop-the-world" garbage collectors, where the application is halted during the garbage collection, create long pauses that affect the application's responsiveness, so concurrent collectors were proposed. Concurrent collectors run in parallel with the application and sometimes stop it for a short phase in the beginning or end of the collection, while on-the-fly collectors stop one thread at a time.

Moreover, servers are required to operate continuously and remain highly responsive to frequent client request. There is a need for concurrently garbage collection algorithms that provide high throughput and high responsiveness for applications with very large live heap data sets.

1.1 Algorithm

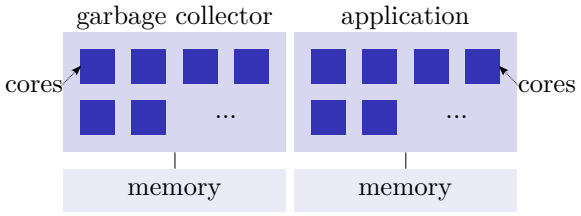

Figure 1: BigGC

[1] http://www.lip6.fr/recherche/team.php?id=740
[2] http://www.lip6.fr
[3] http://www.upmc.fr

Algorithm	Criteria										
	Type						Throughput			Responsiveness	
	parallel	concurrent	compacting	generational	incremental	real-time	read barrier	write barrier	virtual memory operations	STW Phase	Non-blocking
Compactifying[26]	no	yes	yes	yes	no	no	no	no	no	no	spinlock, semaphore
On-the-fly[7]	no	yes	no	no	no	no	no	no	no	no	atomic operations
Garbage First[4]	yes	yes	yes	yes	yes	soft	no	yes-	no	roots scan, final marking, cleanup phase	CAS
Mostly concurrent compaction[18]	~	yes	yes	no	yes	no	no	no	yes	roots, move, stack fix-up	CAS
The Compressor[13]	yes	yes	yes	no	yes	no	no	no	yes	page protection, roots	no
Stopless[19]	yes	yes	yes	yes	yes	soft	yes+cloning-	yes	no	no	lock-free
The Mapping Collector[30]	yes	yes	yes	yes	no	no	no	no	yes	fallback	no synchronisation
Stack scanning[14]	no	yes	~	no	yes		no	yes-	no	no	lock-free with CAS
Hard Real-Time[25]	yes	yes	no	no	yes	hard	no	yes	no	no	
Schism[22]	no	yes	copying	no	no	hard	no	wait-free	no	no	lock-free evacuation
C4[27]	no	yes	yes	yes	yes	no	LVB-	SVB	yes and Quick Release	no	interlock
The Collie[12]	no	yes	yes	no	no	no	LVB-	yes-	no	no	wait-free
NAPS[8]	yes	no	yes	yes	no	no	no	no	no	yes	lock-free

Table 1: Comparison table. The different criteria that were used are presented. Green and yellow cells are considered positive points. Brown and red cells are considered weak points or not answering the principal problem of the BigGC idea.The tilde (~) sign means possible with little modification. The cost of barriers are represented with a plus sign (+) when they have only some instructions that do not have a significant impact on the application performance, with a double plus sign (++) if they are costly in a significant way in application performance and finally with a minus sign (-) if they are filtered/optimized or wait-free

BigGC is a new concurrent garbage collector, that partitions the hardware the garbage collector accesses from the hardware the application does (Fig. 1). The goal is to try to eliminate two major problems of concurrent garbage collectors:

1. the instrumentation of the code application, like memory barriers, that degrade the applications performance
2. the fact that the garbage collector and the application work on the same resources - the heap - which leads to access congestion of the caches, the memory bus, etc.

To achieve partitioning of the heap resource, it replicates the heap in a way that the garbage collector and the application have their own vision of the heap. Moreover, BigGC identifies the *hot*/*cold* parts of the heap to optimize the collection.

The *hot*/*cold* algorithm determines which part of the heap is actively used by the application (*hot* part) and which is not (*cold* part). Then the garbage collector can concurrently collect the *cold* part without bothering the application. The *hot* part is much smaller than the *cold*, so a quick stop-the-world collection is well suited for this part. The distinction of *hot*/*cold* parts minimizes the need for synchronization between the garbage collector and the application.

1.2 Goals

The principal goal of this internship is to implement the new BigGC garbage collector algorithm. The main principle of this goal is to split the heap in two zones:

- *hot* zone: accessed by the application
- *cold* zone: not-accessed by the application

This leads to two basic needs:

- different permission for the heap if it is the garbage collector or the application that accesses the heap, which requiers to replicate the heap
- determine the *hot*/*cold* zones of the heap

1.3 Accomplishments

During the internship the aforementioned goals 1.2 were developed.

The heap replication uses a technique called *double mapping*. The *double map* technique allows to replicate a memory region within the same application. The two different regions are independent concerning the virtual memory system, but the changes made in one are visible to the other.

Two techniques are implemented to determine the *hot*/*cold* zones of the heap. The first one is naïve and uses a simple queue that treats memory of the application with FIFO semantics. The second one uses the better adapted semantics of a LRU list. A Linux module is developed, that piggybacks on the Linux kernel's LRU list to get the necessary information.

1.4 Organization of the report

The report is organized in the following way:

Section 2 presents the state of the art study of concurrent garbage collectors.

Section 3 describes the work that was accomplished and the results of the work.

Section 4 is a study of application locality, of the DaCapo benchmark suite application, that guided development.

Section 5 discusses major difficulties and how they were solved.

Section 6 proposes future work as well as future ideas and proposals for amelioration.

Finally, section 7 concludes on what this work contributes and what remains to do.

2 State of the art

In this section we study the existing art of concurrent garbage collectors. We go over a large number of existing algorithms that try to solve different parts of the problems a concurrent garbage collector algorithm confronts. The Table 1 shows in a compacted form the criteria that were used and how they are associated with the existing art. First, we discuss our base criteria. Later on in this section, follows a more detailed analysis of each algorithm.

2.1 Criteria

The criteria that are used, take into consideration two basic aspects of a garbage collector algorithm, *throughput* and *responsiveness*. Throughput is the average number of operations per time unit. The throughput criteria indicates if the application has a good performance. Responsiveness is the worst response time for an operation. The responsiveness criteria indicate if the application always gives its response in a reasonable time, from the user's perspective.

2.1.1 Performance

Performance criteria that degrade the applications' threads performance due to garbage collector's activity include memory barriers and virtual memory operations.

Memory barrier is a type of instruction which causes a Central Processing Unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Memory barriers are used because most modern CPUs employ performance optimizations that can result in out-of-order execution. This reordering of memory operations normally goes unnoticed within a single thread of execution, but can cause unpredictable behaviour in concurrent programs unless controlled. There are two types of memory barriers: *read barriers* and *write barriers*.

Read barriers are little tasks the application threads execute when reading an address from memory, that help resolve concurrency problems between the applications thread and the garbage collector. The task can be as short as a reference fix-up [12, 27] or long as a relocating or allocating an object [19]. *Read barriers* mark a used or a newly created objects correctly so that, the collector does not consider valid objects as "garbage", while avoiding *costly* synchronisation. This is the case of a moving collector that copies concurrently an object and the mutator tries to use it before all the fix-ups have taken place. In this case, a barrier would prevent the mutator for accessing directly the reference, forcing it to perform the fix-up itself, or follow a forwarding pointer before resuming its business logic.

Write barriers are little tasks the application threads execute when writing a new reference in the heap, that help to make sure that updates during a collection do not get lost. As the collector is concurrently gathering information from the object graph, the mutator can update fields of already marked objects. In this case a *write barrier* would enforce the mutator to inform the collector that there was a modification, and thus, the collector should take into consideration the changes.

Read and *write barriers* affect the mutators' minimum utilization, that is the time that the mutator spends to do the collector's work instead of the application's work. This consumes CPU cycles that could be used to make application progress. A memory barrier *should not* be very long to execute or it can have a significant impact on the applications execution. If the application is very memory intensive a barrier can rapidly become a bottleneck.

Virtual memory operations is a standard way to operate on the virtual memory subsystem of the underlying operating system. A feature of virtual memory is the ability to map file in memory as well as its ability to read protect and write protect individual pages of process memory. The virtual memory operations manipulate the file mappings and the page protection. This means that the operating system can control access to different parts of the address space for each process, and also means that a process can read and/or write protect an area of memory when it wants to ensure that it won't ever read or write to it again. The standard virtual operations [1] we consider are:

Map: Map a virtual page to a physical page.

UnMap: Unmap a virtual page from its associated physical page.

ProtN: Protect a range of virtual pages from read and write access.

UnProt: Remove the protection from a virtual page.

TRAP: Perform a specified routine upon access to a protected virtual page.

DoubleMap: Map one physical page to two different virtual pages.

A garbage collector uses virtual memory operations to read or write protect the pages of the application, so the application can not access these pages unless executing a trap. Forbid the read/write access of pages to an application can be considered a form of *barrier*/synchronisation.

2.1.2 Responsiveness

Responsiveness criteria help the application to be the least impeded by the garbage collector resulting in no wait times for the client. These criteria include stop-the-world phases and the use of locks to achieve cooperation between the collector and the application's threads.

Stop-the-world phases or garbage collectors, stop all the application threads to perform some short synchronisation or to perform the whole collection respectively. The garbage collector algorithm usually pauses the application to perform different operations that ensure correctness and termination. Correctness means that the collector never collects reachable objects and that it will eventually collect all garbage produced by the application. Termination means that a collection cycle will not run eternally due to work introduced by the mutators. The work that the collector perform during the pauses include roots scan, taking a snapshot-at-the-begining or synchronising with the application when the collector's state changes.

Coarser grained synchronisation using locks can effectively impact the responsiveness of the application. When parallel-intensive applications should cooperate with the collector, the use of locks can adversely impact the responsiveness and even the performance of the application.

2.2 Non-moving collectors

Non-moving collectors never relocate objects as part of the garbage collection. This avoids the need to update the references of relocated objects, which usually requires some kind of synchronisation. The drawback is that they can induce fragmentation of the heap. There two types of non-moving collectors, those that try to eliminate fragmentation, and those that don't.

2.2.1 Fragmentation intolerant

On-the-fly[7] Garbage Collection is an on-the-fly garbage collector and its primary goal is to guarantee that it will eventually collect every garbage, while keeping exclusion and synchronisation constraints as weak as possible. It is based on an abstract tri-color algorithm that operates on an objects' graph.

To keep exclusion and synchronisation constraints as weak as possible, the algorithm demands that the collector does not alter the objects' graph in any way other than to append a garbage node. The on-the-fly garbage collector algorithm uses atomic operations and Dijstra semaphores for synchronisation.

The overhead on the mutator is negligible and the mutator's activity does not impair the collector from identifying and collecting garbage.

The stack scanning[14] algorithm develops methods for concurrent, incremental and lock-free stack scanning garbage collectors. The algorithm tries to achieve simultaneous scanning of the stack by the collector and the mutator while preserving lock-freedom, as well as handling interaction between managed and unmanaged code.

The collector uses a return-barrier technique that is cleverly set up to be executed in rare cases. This barrier makes the mutator perform some work on behalf of the garbage collector if and only if the garbage collector has not performed it yet. To handle parameters passed by reference, the collector uses also *write barrier*. Handshakes between the collector and the mutator provide synchronisation.

The impact of the *write barrier* is insignificant, as it is employed only when the garbage collector is operating concurrently on the same part of the stack as the mutator operates on. This algorithm achieves high responsiveness to the microsecond level. It also provides support for programs that employ fine-synchronization to avoid locks.

2.2.2 Fragmentation tolerant

The Mapping Collector[30] is a generational, parallel, concurrent (or stop-the-world), non-moving, nearly single-phase compacting garbage collector. The Mapping Collector (MC) goal is to achieve perfect compaction, without any object moving, while preserving order between objects before and after the compacting phase.

The MC employs virtual memory operations, like unmapping and remmaping, as primary technique to achieve compaction, as it exploits the widely-observed statistical property that unreachable objects tend to cluster together and form contiguous dead regions in the heap. The MC remaps the free space into a contiguous region in a newly allocated area in virtual memory. As the MC does not access live objects, it can run concurrently with the application without the need for any synchronisation.

If the collector fails to reclaim a sufficient amount of free space, then it falls back to perfect compaction using a state-of-the-art compacting algorithm (cf. The Compressor[13]).

The MC increases throughput and scalability while drastically reducing pause times (up to 70% compared to The Compressor[13]), while adding a space overhead below 6%. The MC also speeds up application execution time (up to 5.9). Finally, the fall-back rate to the perfect compacting algorithm is about 6.5%.

Schism[22] is a concurrent and real-time garbage collector, fragmentation tolerant and guarantees time-and-space worst-case bounds. The main goal of Schism is to beat memory fragmentation for real-time systems.

The Schism algorithm allocates object by fragments of 64 bytes, as it is the size of most objects, and for bigger objects, it uses a linked list of fragments. That way Schism copes with external fragmentation when running in small heaps. The fragments are always of fixed size and never move thus achieving constant-time heap access and constant space-bounds for internal fragmentation. Arrays are treated as large objects and are represented by a spine, containing pointers to a set of array fragments.

Schism uses wait-free barriers for reading and writing the array spines. The allocation of spines can also be made lock-free.

Schism runs 1.6 to 2.5 times slower than the reference implementation of HotSpot, but it offers appreciable scalability on modern multi-cores. Schism provides different predictability levels, hence making it a hard real-time garbage collection solution.

2.3 Moving collectors

Moving collectors are a type of collector that moves objects during the garbage collection phase. The move phase can be done either by copying objects or by using some virtual memory operations without copying. The main interest of moving collectors is to compact the heap and avoid fragmentation. The compaction leads to better locality and to more efficient use of the memory address space. Also it makes it possible to use a very efficient algorithm for allocation that only has to update the pointer to the end of the last allocated object to allocate a new one. There are two types of moving collectors, those that relocate objects by pausing the mutators and those that perform the relocation concurrently with the mutators.

2.3.1 Pausing during compaction

Garbage First[4] 's goals are high throughput as well as soft real-time constraints. Garbage First has high probability of not violating the soft real-time constraints, because it collects the most interesting (efficient) part of the heap. The most interesting parts of the heap is based on an accurate model of the cost of collecting equal sized heap regions, so the algorithm can conclude which regions can be collected within the given pause time limit.

Garbage First employs stop-the-world phases, evacuation pauses and *write barriers*. The stop-the-world phases perform the roots scanning, the final marking, the evacuation and the clean-up phase. The evacuation pauses, that allow compaction, are optimised to be performed in incremental stop-the-world phases in parallel, using work-stealing techniques. Garbage First also uses a *write barrier* to track concurrent updates. The *write barrier* is used to keep up-to-date *sets* that indicates all locations of live objects. The *write barrier* is optimised using filtering techniques that can filter up to 81% of pointer writes. If the write creates a pointer in the same heap region, then it needs not be recorded in a *set*. At last, Garbage First can switch between fully and partially young modes for better efficiency. A generational execution starts in fully-young mode. After a marking pass is complete, Garbage First switches to partially-young mode, to collect attractive non-young regions identified by the marking. If the efficiency of the partial collections declines to the efficiency of fully-young collections, it switches back to fully-young mode.

Garbage First offers greatly reduces pause times that can be tuned by the user. The user soft real-time goals are violated less than 5% of the time. Moreover, it scales quite well up to 7 CPUs, while having a space overhead that is negligible.

In mostly concurrent compaction[18] an incremental, concurrent compaction algorithm is studied, with main goal to reduce the pause time created by compaction.

The algorithm uses many techniques such as virtual memory page protection techniques, mutator traps, heap sections and fix-up phases. A fix-up phase updates all references to the moved objects after compaction. If an application thread accesses a protected page, it executes a trap routine, which fixes that page and then unprotects it, before continuing to execute. The algorithm also gathers heap section information, during the sweep phase, that allows to select the most promising heap sections for compaction. The compaction is entirely performed in a stop-the-world manner, and it is based on existing moving algorithm.

The pause time is bounded. The cost of page protection varied from 0.01 to 0.13 ms per each page access violation that introduces a significant overhead. The average overhead for fix-up is 200ms which is also quite significant. The MMU very high up to 90%.

2.3.2 Concurrent during compaction

The Compactifying[26] algorithm compacts and relocates cells concurrently with the mutator.

The algorithm uses "spin-locks" and Dijkstra semaphores[6] for synchronisation. The spin-lock avoid manipulations of the same variable in the same time, between the garbage collector and the mutator. The spin-lock can be considered as a *read barrier*.

The overhead on the mutator is negligible to the extent that the mutator never needs to wait the garbage collector other than short periods of time. The mutator's activity does not impair the collector from identifying and collecting garbage.

C4[27] is a a multi-generational, continuously concurrent, always compacting garbage collector. The main goal is to eliminate any pauses during compaction. C4 supports simultaneous generational concurrency using multiple independently running instances to simultaneously collect all generations. The simultaneous multi-generation operation, limits the cross-generational synchronisation.

The C4 uses *read barriers* and *write barrier* as well as virtual memory operations. The *read barrier* imposes a set of invariants on every object reference value as it is loaded from memory. The invariants provide safety for mark and access. Using a self-healing[4] technique, it avoids repeated triggers on the same loaded reference, dramatically reducing the dynamic occurrence of read barriers, leading to efficient and predictable fast path test execution. The *write barrier* only keeps track of young-to-old generation references.

C4 uses interlocks to solve synchronisation between the two simultaneous collections. An interlock, briefly halts one of the collectors, to provide safe access, at page granularity.

The C4 virtual memory operations are based on page mappings and unmappings. To achieve high performance the C4 implements a new virtual memory subsystem for the operation system. The property of this new subsystem is the ability to change the protection of pages without causing a Translation Lookaside Buffer (TLB) invalidation.

The virtual memory subsystem can sustain virtual mapping manipulation rate providing high throughput. It has very good scalability properties as well as two-orders-of-magnitude improvement in sustainable worst case response times, compared to HotSpot's ParallelGC.

The Collie[12] is a fully concurrent compacting collector, that focuses on individual object relocation. The collector uses compaction as the primary mean of reclaiming unused memory. The Collie uses referrer sets to identify the precise set of all object references that point to a object's location.

[4]Self-healing means that it heals source memory location by atomically storing a copy of the reference back to the source location.

The Collie uses *read barriers*, *write barriers*, virtual memory operations as well as transactional memory techniques. The *read barrier* intercepts all reference loads from the heap, helping the concurrent tracer and the concurrent compactor. Aborting *read barriers* reduce the costs of the *read barriers* and they use transactional memory techniques.

The Collie uses virtual memory operations to create a mirrored *to-space*, so it can perform zero-copy, non compacting, transplantation, on a page granularity. The transplantation is achieved in a wait-free, bounded constant time, using a cheap *read barrier*.

The *write barrier* intercepts all stores of references to the heap, identifying whether an object should be marked as "non-individually transplantable".

The MMU beats all metrics compared to Pauseless [2] . The Collie improves also the overall throughput but the focus was not on improving throughput. The throughput improvement is a side effect of the lower read barrier triggering rates and the reduced mutator work on triggered barriers. The throughput is 1.09 to 1.15 better than the Pauseless [2].

The Compressor[13] is a concurrent, parallel and incremental garbage collection algorithm, which compacts the entire heap to a single condensed area.

The Compressor uses virtual memory operations as well as markbit and offset vectors to achieve full compaction. The collector handles the compaction while the mutators are running concurrently, needing only a stop-the-world phase to update the roots and protect the pages that require update. Using virtual memory operations it also avoids the use of forwarding pointers by trapping the mutators. The move and the references fix-up is executed in parallel with no synchronisation, when the mutator threads get trapped.

The Compressor improves locality and cache consciousness of the collector because the moving phase uses only the two markbit vectors and do not touch the heap. It reduces pause times significantly, allowing acceptable runs on large heaps, requiring only a single heap pass while achieving full compaction. f The Compressor achieves full compaction, having only a space overhead of ˜4%.

Stopless[19] is a concurrent, real-time, parallel, compacting garbage collector. The principal goal is to have extremely short pause time and have lock-freedom. Stopless provides (soft) real-time with high responsiveness, preserving lock-freedom, supporting atomic operations, controlling fragmentation by compaction on stock hardware. It performs object copying with a temporary wide object, employing also a forwarding pointer. A wide object contains for each field the actual location of the field: original location, temporary location or final location.

Stopless uses *read barriers* and *write barriers* for synchronisation. To access wide objects, Stopless employs a synchronisation method that uses *read barriers*. A novel code cloning technique drastically reduces the barrier's effect, up to a factor of 2. The code cloning technique is a compiler assisted technique, that has two copies of the code, one for the fast path and one for the slow path. The *read barrier's* primary role is to inform where the up-to-date version of the associated field of an object is.

Stopless uses write barriers to make sure that updates do not get lost between the different copies of an object.

Lock-freedom is provided with a lock-free virtual mark-stack, a lock-free work-stealing mechanism and a termination condition that does not block any mutator threads.

The responsiveness is two orders of magnitude, a factor of 100, better than previously published systems, resulting in less than tens of microseconds. The collector offers an acceptable scalability and throughput.

2.4 Multicore garbage collectors

NAPS[8] is a stop-the-world, generational, copying and compacting garbage collection algorithm based on the Parallel Scavenge, with focus on scalability on NUMA machines.

NUMA-aware Parallel Scavenge (NAPS) greatly impacts scalability by distributing live objects in a balanced way between the NUMA nodes and completely avoids locking during the parallel phase of the collection. It also simplifying the synchronisation protocol of the garbage collector's threads. The NAPS algorithm employs lock-free structures for the collector task queue. It uses a lazy garbage collector parking technique that completely removes the monitor's lock.

NAPS scales well with the number of cores and performance continues to increase up to 48 cores while improves application time by up to 28%. The improvement on the application time is a side effect of the memory balancing of the algorithm. NAPS reduces individual pause times by up to 2.8 times improving the responsiveness of the application.

3 Accomplishments and results

A *double map* technique that allows to replicate a memory region within the same application constitutes the first part of my work. This technique is implanted in the HotSpot JVM to replicate the heap.

Then two techniques are implemented to determine the *hot*/*cold* zones of the heap. The first one is naïve and uses a simple FIFO queue. This implementation is actually implemented in the HotSpot JVM code.

The second one uses a LRU list. A Linux module is developed, that piggybacks on the Linux kernel's swapper LRU list to get the necessary information. The Linux module nearly completed, meaning that is functionnal in a proof-of-concept way, and thus the integration with the HotSpot's JVM is not there yet.

Finally, a study of applications' locality was conducted to better understand how to identify the *hot*/*cold* zones. The study is based on the FIFO queue implementation.

3.1 Double Map

As noted in the the Abstract and the 1 Introduction section, the aim is to find and implement an algorithm to separate the heap into two parts, one with frequently accessed objects, named the *hot* space, and the other with non-frequently accessed objects, named the *cold* space.

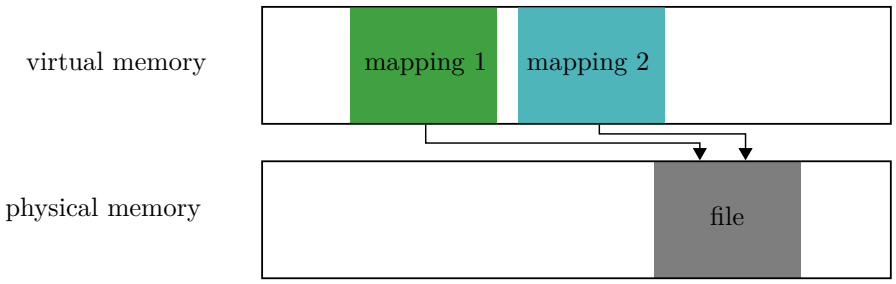

Figure 2: Visualization of the *double mapping*. Here the mapping 1 and the mapping 2 are pointing into the same physical memory, but are two distinct virtual memory regions, that do not overlap.

The *hot* space must be accessible by the application while the *cold* one should be protected from any access from the application. Thus, the garbage collector can concurrently perform the collection work in the *cold* space. The problem that arises from the protection of the *cold* space is that the garbage collector *must* be able to access this space, while the application should not. So, a solution is needed that will provide different protections for the same physical memory range.

A technique, hereafter named *double mapping*, was used to achieve the desired result. Double map means to associate a physical memory address range to two locations in the virtual memory of the applications. It is like having two distinct views of the same object, or having two proxies for the same object (Fig. 2). An example of *double mapping* is when two distinct processes map a file, with `mmap(2)`, in their virtual address space with the flag `MAP_SHARED`. Updates to the mapping are visible to other processes that map this file, and are carried through to the

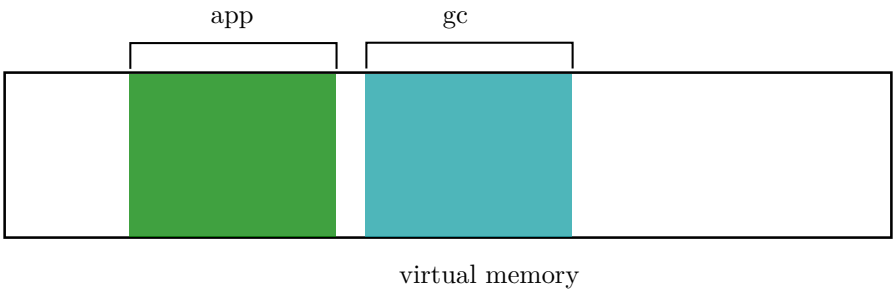

Figure 3: Visualization of the *double mapping* used by the garbage collector.

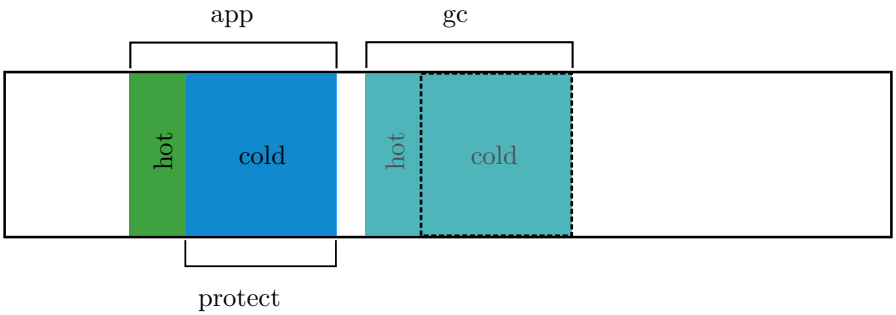

Figure 4: Visualization of the protection mechanism that leverages the use of the *double mapping*.

underlying file, the back-store. This way the processes can access the same physical memory from two different virtual memory addresses.

3.1.1 Why *double map* the heap

The resource that the garbage collector operates on is the heap of the application and to have different protections for the heap, the garbage collector has to use the *double map* technique on the heap, from within the *same* process.

The goal is to have two separate virtual memory ranges, that can be independently used from the application and the garbage collector, as seen in Figure 3. This way the heap can be double mapped and give the first mapped address to the application, and keep the second one for the garbage collector. That way the garbage collector can protect (with `mprotect(2)`) the part of the first address range and thus prevent the application from accessing that part of the heap, while the garbage collector will access the same (physical) memory region via the second mapped virtual address during a concurrent garbage collector, as shown in Figure 4.

3.1.2 How does double mapping work

Double mapping a file into memory is straightforward: (1) Create a file, (2) truncate the file to the desired size, (3) map the file descriptor, with the shared flag (`MAP_SHARED`), into memory twice. The result is two shared virtual mappings pointing to the same file. The shared mappings are placed into different virtual address regions and do not overlap (Fig. 2). These two regions can now be manipulated separately with `mprotect(2)`, hence achieving the desired effect of

separating the heap the application uses, from the heap the garbage collector performs collection on. The file used as the back-store can be `unlink(2)`ed immediately after the `open(2)` call and `close(2)`d immediately after the double `mmap(2)`call.

3.1.3 Something to consider

The `mmap(2)`ed files have as back-store a file to disk. To have the changes performed to the mappings visible to every mapping with the `MAP_SHARED` flag, the system needs to propagate the changes back to disk. Thus, as we *double map* the heap, propagating the changes back to disk degrades application performance.

To overcome this limitation, the file that serves as a back-store to the heap can be created in a `tmpfs` (cf. [15]) mount point. This means that the back-store of the file is in memory and thus there is no performance impact[5].

The `mmap(2)`ed files are considered as *page cache* by Linux. Since `tmpfs` lives completely in the *page cache* and on swap, all `tmpfs` pages currently in memory, will also show up as cached. Pages in the *page cache* are managed differently from anonymous pages. Linux, if under memory pressure, first tries to swap out *page cache* pages before the anonymous ones. The potential impact to the application is that its heap memory has more chances to be swapped out sooner than other processes' memory that is anonymous, and hence degrade performance. A possible solution to the swap out problem is to use `mlock(2)` or `madvice(2)` system calls.

3.2 Hot Page Queue

3.2.1 Page protection mechanism

The space of the heap is divided in two virtual spaces, the *hot space* and the *cold space*. The *hot* space should contain pages that are frequently used and pages that are in constant use by the application. The *cold* space contains the rest of the pages. *Cold* pages are promoted to the *hot* space when they are accessed. At this time, the coldest page from the *hot* space moves to the *cold* space.

The page protection mechanism protects the *cold* pages, using the `mprotect(2)` system call, so that the application can not access them without provoking a segment violation error. This error will provoke a `SIGSEGV` signal to be sent to the JVM. This signal is handled by a signal handler that traps the thread accessing the *cold* page and promotes the page to the *hot* space and/or informs the garbage collection system.

The *hot*/*cold* space architecture liberates the garbage collection system of any need of explicit synchronization with the application; thus the garbage collector can concurrently operate on the *cold* space. If the application ever tries to touch a protected page then the trap will cooperate with the garbage collector. Cooperating either means informing the garbage collector of the fact that a mutator tries to access a *cold* page, or either to perform some garbage collection work on the behalf of the garbage collection by the mutator.

The signal handler is quite an expensive operation and thus the application should not access very often the *cold* space. To amortise and minimize the cost of the signal handler some techniques are explored. Like the FIFO queue or the LRU queue that are discussed in the next sections.

[5]The use of a `tmpfs` mount point can have other side effects. As the contents of the `tmpfs` are occupying real memory, if the system is under memory pressure, this can lead to strange system behaviour, if the system can not free enough memory.

Commands		Results	
`register`	register a `pid`	`clients/`	the directory containing the clients that are registered
`unregister`	unregister a `pid`		
`update`	demand an update of the LRU list	`clients/<pid>`	the file containing the last LRU list that was updated due to an `update` request
`enabled`	enable/disable the module		

Table 2: Description of Linux's module `sysfs` interface, in the `/sys/module/<module>/state` directory.

3.2.2 FIFO Queue

To separate the heap in *hot* and *cold* spaces, there is a need to find the most recently used pages of the application. The first naïve and simplistic approach is to have a fixed size FIFO queue containing the *hot* pages. Each time the page protection mechanism is engaged, this means that application tries to access a *cold* page. The mechanism appends the *cold* page in the hot page queue, hence the page is promoted to the *hot* space. If the hot page queue is full, then the first page is removed from the queue and protected, so the application can not access it.

In order to avoid contention on a global FIFO queue each thread has its own, independent, FIFO queue. This approach is reasonable because it takes into consideration the application's locality principle; the threads access more often their own data and rarely global data. Moreover, a page that was recently accessed by a thread is more likely to be accessed again in the near future by the same thread. So, the size of the FIFO queue expresses the number of pages in each thread's queue.

3.2.3 LRU list

A more clever and advanced approach is to use an LRU list of fixed size to keep track of the *hot* pages. The Linux operating system already has a LRU list that is used by the swapper. So a clever way to manage the *hot* space is to piggyback on the Linux kernel to extract this information.

A Linux module that extracts this information was developed. The module presents an interface representation in the *sysfs* (cf. [17]) file system that an userspace application can use (Table 2). The module offers on demand support to the application for LRU information.

When the garbage collector begins a collection cycle, it asks the module to update the LRU list. Then it reads this list and it concludes the *hot*/*cold* spaces, protects the corresponding spaces and starts a stop-the-world collection of the *hot* space. After the stop-the-world collection it collects the *cold* space concurrently with the application.

`control group (cgroup)` A very promising feature of the Linux kernel is the `cgroup` memory resource controller [16]. The controller allows to control different resources for a group of applications. The most interesting feature however is the fact that the applications in a cgroup have their *own* LRU list. The separate LRU list facilitates the development of the Linux module, as it has to browse a smaller list and do less checks for the owner of the page, as all pages are coming from one application only.

The cgroup is also promising for a future shared garbage collector implementation (cf. 6).

4 Study/Analysis of application locality

The DaCapo benchmark suite[3] (cf Glossary and A) is intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It consists of a set of open source, real world applications with non-trivial memory loads.

The DaCapo benchmark suite was used to study the impact of the hot page queue (cf. 3.2) in application time as well as to conclude on the size of the queue that does no degrades a lot application performance. The results below use the FIFO queue only, because the Linux module (cf. 3.2.3) is not yet completed.

All the tests were conducted in a 48-core AMD Magny-Cours NUMA machine with 8 memory nodes and a total memory size of 32GB.

The page protection mechanism (cf. 3.2.1) was modified to track how many times it was engaged for different sizes of the hot page queue. Figure 5 presents the results. The benchmark's applications were run with 32GB of heap. The average page faults are application depended. The average number of threads per application is about 60, the average page faults at the 1000 page threshold is 85349 faults and the average variance at 1000 page threshold is $\sigma = 1233754.83640$.

The curve is logarithmic and we perceive that a larger threshold of for the queue, results in fewer faults. This means that there are more *hot* pages available at the same time and thus there is no need to provoke a page fault.

Starting from 1000 pages per thread, the impact of the queue on the application performance stabilizes. This shows us that the application locality principle can effectively taken into consideration for the *hot* page queue algorithm and that it will not seriously degrade application performance. The 1000 page threshold was chosen as the best size for the hot page queue.

The other test that we conducted was to see how many times a page goes in the hot page queue. The page protection mechanism (cf. 3.2.1) was modified to track how many times each page gets in the queue. Figure 6 presents the results. The benchmark's applications were run with different heap sizes, but with a fixed page threshold of 1000 pages.

The results show that the 100 hottest pages get in average 4058 times in the queue ($\sigma = 36456.473$). This number depends on the size of the heap and the way the application behaves. The average times all pages get in the hot page queue for all application is around 2.5.

Figure 7 presents a comparative graph for the DaCapo benchmark suite applications of the *hotness* of the 100/500/1000 hottest pages. The *hotness* metric represents the number of times a page has got in the queue. The 100 hottest pages are much hotter than the rest of the pages. This provides meaningful information that the

LRU algorithm can take advantage of.

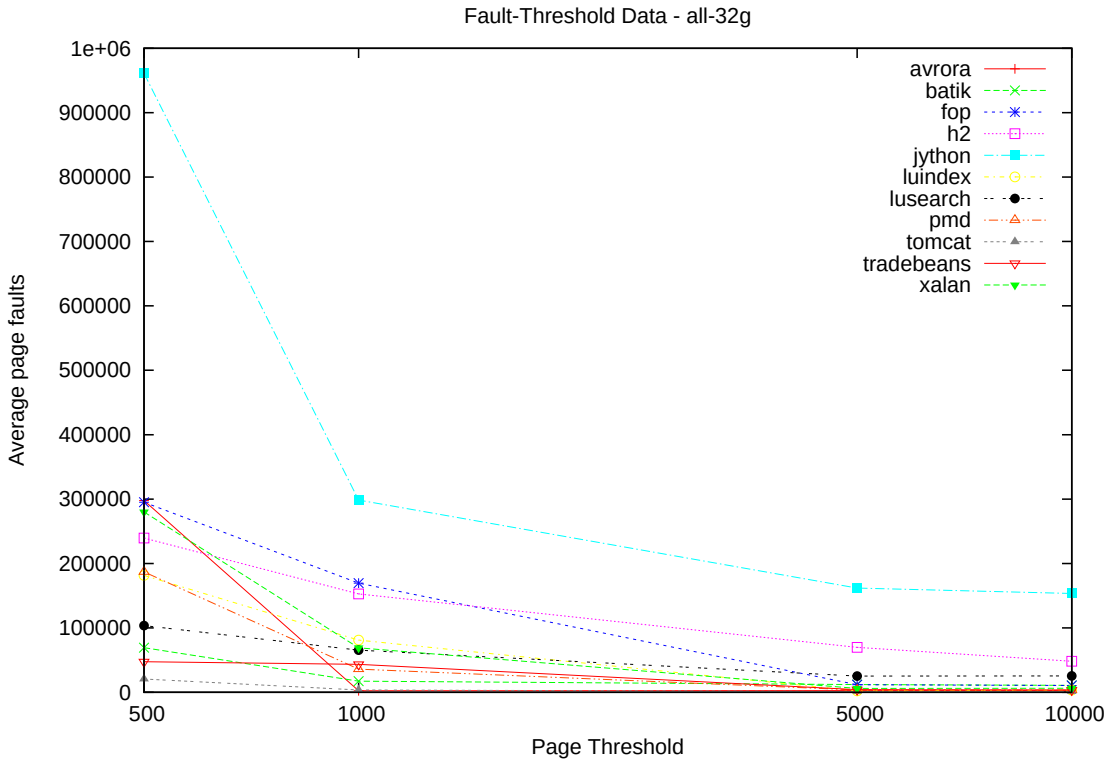

Figure 5: DaCapo benchmark suite run with different size thresholds of hot page queue (cf. 3.2). The execution of each benchmark application was done with a heap size of 32GB. From the 1000 page threshold per thread, the average number of times a page gets in and out of the queue stabilizes.

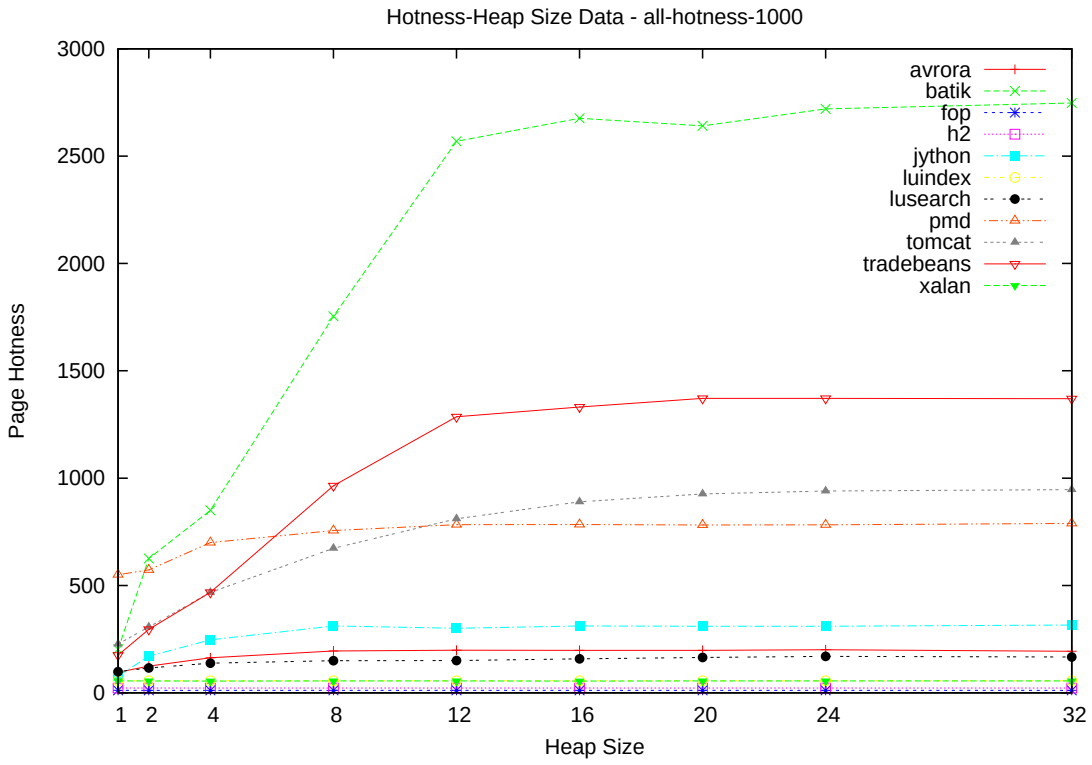

Figure 6: This graph shows the number of times the 100 most hot pages get in the hot page queue with a threshold of 1000 pages (cf. 3.2). The page hotness depends on the application and on the size of the heap.

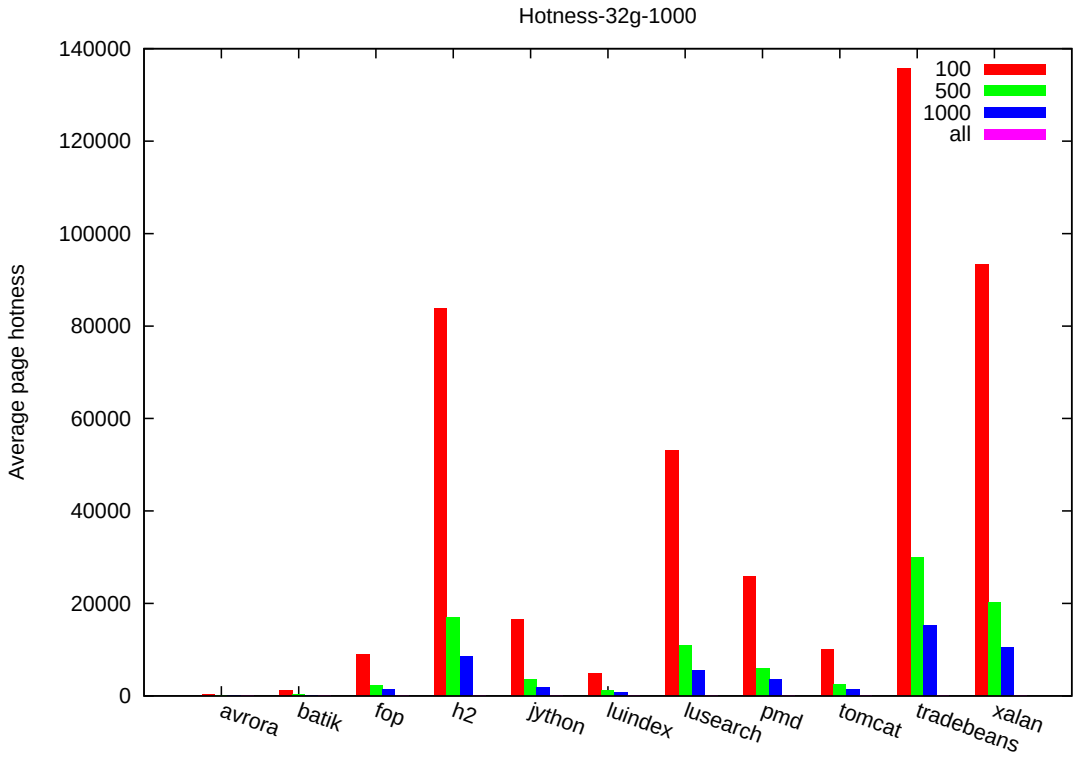

Figure 7: This graph shows the number of times the 100 most hot pages get in the hot page queue with a threshold of 1000 pages (cf. 3.2), for a heap of 32GB. It is clear that the 100 hottest pages are *much* hotter than the rest of the pages.

5 Difficulties

The *double mapping* technique is rarely used this way. The principal use of *double mapping* is to share memory between processes, as an Inter-Process Communication (IPC) mechanism. The IPC mechanism is often limited to a size of some megabytes, which does not respond to the needs of a big data garbage collector, like BigGC. The need of a robust mechanism was apparent and so, the most promising mechanism already existing in Linux, was the `mmap(2)` system call. `mmap(2)` is robust, fast (and lazy) and has already the support to share memory between processes as well as to map a file in memory.

Figure 8: Visualization of the alignment choices of the heap in JVM. The upper rectangle designates the memory and the green rectangle designates the heap allocation. The alignment of the heap start is not one that JVM likes. The result is to attempt to reserve a little more memory from the same starting address and add a little offset inside that region that is correctly aligned for the needs of the JVM.

The biggest problem was to implant the *double mapping* code in the HotSpot JVM code base. This was quite a difficult task, because the way the memory reservation works in the HotSpot. The process is not very clear and complicated. The biggest problem was the strong requirements in terms of alignment of the heap region the JVM imposes. This results in many attempts of memory reservation by the JVM, trying to achieve the correct alignment (Fig. 8). When the allocated memory region for the heap is not aligned to specific boundaries, the JVM tries to fix the situation by calculating an offset. Then it allocates again the same amount of memory plus the calculated offset and asks for a `mmap(2)`ed allocation from the same starting address as the last one, using the `MAP_FIXED` flag. The solution was to keep track of the allocations and `munmap(2)` the previous allocations.

Another problematic feature of the JVM is the way it handles memory allocation as well as tracking it, with `ReservedSpaces`. `ReservedSpaces` is a cluster of classes that track the regions of memory allocated for different purposes by the JVM. One subclass of the `ReservedSpace`, the `ReservedHeapSpace` is of particular interest because it describes the range of the heap. However, there are different types of heaps in the JVM. There is a heap for the Just-In-Time (JIT) compiler, one for the JAVA application etc. A way to indicate to the `mmap(2)` call that a *double map* should take place was needed. This step took a considerable part of the internship time.

Once this obstacle tackled down, there was the Page Protection Mechanism (cf. 3.2.1) that complicated even more the already not evident code of the heap *double map*. The Page Protection Mechanism uses a `SIGSEGV` signal handler to trap the threads that try to access a protected page. The handling code, should find if the page that is accessed belongs to the heap region. Since the

application, the garbage collector threads and the Virtual Machine (VM) threads access the heap the signal handler is called by different threads, in different moments, like while the application is running or when the garbage collector is running, the implementation that works in all the above cases was not evident.

6 Future work

With the map technique coupled with the *hot* space management the garbage collector can be shared between many JVM in the same machine. All the *cold* spaces can be collected concurrently while the applications are running. The garbage collection can be done in a clever way that does not invalidate the caches and thus does not destroy the applications' locality. Especially in a NUMA machine these details can have an enormous gain in the application performance.

7 Conclusion

BigGC actually starts to shape into a modern concurrent garbage collector. The first milestones are now put in place. Starting from the initial objectives the replication of the heap as well as the identification of the *hot*/*cold* zones, both are now accomplished. What remains to be done is to:

- complete the Linux module
- modify HotSpot JVM to communicate with the module
- modify the actual garbage collector algorithm in HotSpot so that it performs garbage collection on the replicated heap
- advance in novel directions, that no garbage collector has ever been

A DaCapo benchmark suite applications

avrora	simulates a number of programs run on a grid of AVR microcontrollers
batik	produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
h2	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark
jython	inteprets a the pybench Python benchmark
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
tomcat	runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages
tradebeans	runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database
xalan	transforms XML documents into HTML

Glossary

CAS In computer science, compare-and-swap (CAS) is an atomic instruction used in multithreading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail. The result of the operation must indicate whether it performed the substitution; this can be done either with a simple Boolean response (this variant is often called compare-and-set), or by returning the value read from the memory location (not the value written to it).[31]. 28

double map to map the same physical memory of an application in two different virtual addresses, using the `mmap(2)`or `shmget(2)`, `shmat(2)` calls. 6, 9, 15–17, 23

garbage collector In computer science, garbage collection (GC) is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called finalization. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming. [32] 2, 4, 6, 8–10, 12, 15–18, 23–26, 28

heap in programming, it refers to a common pool of memory that is available to the program. The management of the heap is either done by the applications themselves, allocating and deallocating memory as required, or by the operating system or other system program [5]. 15

map to map a virtual page to a physical page, usually using the `mmap(2)`system call. 9

memory barrier A memory barrier is a type of instruction which causes a CPU or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier. (cf. CAS) [33]. 6, 8

MMU the time a mutator spends to do the garbage collector's work instead of the application's work.. 8, 12, 13

prot to disassociate a virtual page from its associated physical page, using the `mprotect(2)` system call. 9

sysfs sysfs is a virtual file system provided by Linux. sysfs provides a set of virtual files by exporting information about various kernel subsystems, hardware devices and associated device drivers from the kernel's device model to user space. In addition to providing information about various devices and kernel subsystems, exported virtual files are also used for their configuring. sysfs is similar to the sysctl mechanism found in BSD systems, but implemented as a file system instead of a separate mechanism.[34]. 18

trap to disassociate a virtual page from its associated physical page. 9, 23

unmap to disassociate a virtual page from its associated physical page, usually using the `munmap(2)` system call. 9

unprot to disassociate a virtual page from its associated physical page, using the `mprotect(2)` system call. 9

Acronyms

cgroup control group. 18

CPU Central Processing Unit. 8, 28

FIFO First-In First-Out. 3, 6, 15, 18, 19

IPC Inter-Process Communication. 23

JIT Just-In-Time. 23

JVM Java Virtual Machine. 2, 15, 17, 23, 25, 26

LRU Least Recently Used. 3, 6, 15, 17–19

NAPS NUMA-aware Parallel Scavenge. 5, 14

TLB Translation Lookaside Buffer. 12

VM Virtual Machine. 24

References

[1] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 96–107, New York, NY, USA, 1991. ACM.

[2] Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 46–56, New York, NY, USA, 2005. ACM.

[3] DaCapo. the DaCapo benchmark suite. `http://www.dacapobench.org/`, Feb 2012.

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[5] The Free Dictionary. Dynamic memory allocation. `http://encyclopedia2.thefreedictionary.com/heap`.

[6] Edsger W. Dijkstra. The structure of the “the”-multiprogramming system. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 10.1–10.6, New York, NY, USA, 1967. ACM.

[7] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[8] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 229–240, New York, NY, USA, 2013. ACM.

[9] Antony L. Hosking. Portable, mostly-concurrent, mostly-copying garbage collection for multi-processors. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 40–51, New York, NY, USA, 2006. ACM.

[10] Nikolay Igotti. Double mapping of memory regions on unix. `https://blogs.oracle.com/nike/entry/double_mapping_of_memory_regions`, Jul 2007.

[11] Balaji Iyengar, Edward Gehringer, Michael Wolf, and Karthikeyan Manivannan. Scalable concurrent and parallel mark. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 61–72, New York, NY, USA, 2012. ACM.

[12] Balaji Iyengar, Gil Tene, Michael Wolf, and Edward Gehringer. The collie: A wait-free compacting collector. In *Proceedings of the 2012 International Symposium on Memory Management*, ISMM '12, pages 85–96, New York, NY, USA, 2012. ACM.

[13] Haim Kermany and Erez Petrank. The compressor: Concurrent, incremental, and parallel compaction. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 354–363, New York, NY, USA, 2006. ACM.

[14] Gabriel Kliot, Erez Petrank, and Bjarne Steensgaard. A lock-free, concurrent, and incremental stack scanning mechanism for garbage collectors. *SIGOPS Oper. Syst. Rev.*, 43(3):3–13, July 2009.

[15] Linux. tmpfs. `https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt`, March 2010.

[16] Linux. Documentation/cgroups/memory.txt. `https://www.kernel.org/doc/Documentation/cgroups/memory.txt`, March 2011.

[17] Linux. sysfs - the filesystem for exporting kernel objects. `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`, August 2011.

[18] Yoav Ossia, Ori Ben-Yitzhak, and Marc Segal. Mostly concurrent compaction for mark-sweep gc. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 25–36, New York, NY, USA, 2004. ACM.

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

[20] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 33–44, New York, NY, USA, 2008. ACM.

[21] Filip Pizlo and Jan Vitek. Memory management for real-time java: State of the art. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 248–254, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. *SIGPLAN Not.*, 45(6):146–159, June 2010.

[23] Ivan Seelnon. When I'm Atatata. `https://www.youtube.com/watch?v=PMO2gjopqqk`, Jul 2012.

[24] Fridtjof Siebert. Limits of parallel marking garbage collection. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 21–29, New York, NY, USA, 2008. ACM.

[25] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 11–20, New York, NY, USA, 2010. ACM.

[26] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, September 1975.

[27] Gil Tene, Balaji Iyengar, and Michael Wolf. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[28] Linus Torvalds. double mmap calls. `https://groups.google.com/forum/#!topic/comp.os.linux.development.system/Prx7ExCzsv4`, Jan 2001.

[29] Martin T. Vechev and David F. Bacon. Write barrier elision for concurrent garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 13–24, New York, NY, USA, 2004. ACM.

[30] Michal Wegiel and Chandra Krintz. The mapping collector: Virtual memory support for generational, parallel, and concurrent compaction. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 91–102, New York, NY, USA, 2008. ACM.

[31] Wikipedia. Compare-and-swap. `https://en.wikipedia.org/wiki/Compare-and-swap`.

[32] Wikipedia. Garbage collection. `https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)`.

[33] Wikipedia. Memory barrier. `https://en.wikipedia.org/wiki/Memory_barrier`.

[34] Wikipedia. sysfs. `https://en.wikipedia.org/wiki/Sysfs`.