A study of the scalability of stop-the-world garbage collectors on multicores

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro

A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores *

Lokesh Gidra, Gaël Thomas, Julien Sopena and Marc Shapiro

LIP6-INRIA
Université Pierre et Marie Curie
4 place Jussieu, Paris, France
firstname.lastname@lip6.fr

Abstract

Large-scale multicore architectures create new challenges for garbage collectors (GCs). In particular, throughput-oriented stop-the-world algorithms demonstrate good performance with a small number of cores, but have been shown to degrade badly beyond approximately 8 cores on a 48-core with OpenJDK 7. This negative result raises the question whether the stop-the-world design has intrinsic limitations that would require a radically different approach. Our study suggests that the answer is no, and that there is no compelling scalability reason to discard the existing highly-optimised throughput-oriented GC code on contemporary hardware. This paper studies the default throughput-oriented garbage collector of OpenJDK 7, called Parallel Scavenge. We identify its bottlenecks, and show how to eliminate them using well-established parallel programming techniques. On the SPECjbb2005, SPECjvm2008 and DaCapo 9.12 benchmarks, the improved GC matches the performance of Parallel Scavenge at low core count, but scales well, up to 48 cores.

Categories and Subject Descriptors D.4.2 [*Software*]: Garbage collection

General Terms Experimentation, Performance

Keywords Garbage collection; NUMA; multicore

1. Introduction

Contemporary hardware with large memories and high core counts creates new challenges for garbage collectors (GCs). To achieve the best possible performance, throughput-oriented GCs traditionally use a stop-the-world design, in which the collector pauses the application threads during the whole collection to prevent concurrent memory accesses from the application. Stop-the-world is the simplest design and is relatively easy to combine with advanced techniques [12]. In contrast, a concurrent GC does not pause the application, but is much more complex: as it runs in parallel with application code, it requires complicated fine-grain synchronisation with the GC and costly code instrumentation to intercept reference mutations [1, 7, 16, 24, 27, 28].

According to some authors, for instance Iyengar et al. [11], stop-the-world collectors are not adequate on large machines because their pause time increases with available memory. This reasoning ignores the counter-balancing effect that large machines also come with many cores, translating to more parallelism. If efficiently exploited, this parallelism should reduce the pause time. However, our previous study [10] shows that the performance of the stop-the-world GCs available in OpenJDK 7 degrades beyond about 8 GC threads, taking over one-third of total application time at maximum parallelism (one GC thread for each of 48 hardware cores). This performance issue questions the design of stop-the-world collectors: are they intrinsically unable to scale with the number of cores, and thus, is it now time to pay the price of concurrent collectors to collect large heaps?

The result of our study suggests that the answer is no, at least on an off-the-shelf 48-core NUMA machine.[1] Our study is based on Parallel Scavenge [21], the stop-the-world throughput-oriented collector used by default in OpenJDK 7. We show that its bad performance with a lot of cores is not intrinsic to its design, but is due to two bottlenecks, which we correct using well-established parallel programming techniques:

- **Lack of NUMA-awareness.** Many objects are allocated on a single node, overloading it, during execution of both the application and the GC code. To correct this issue, we compare three different approaches to balancing memory access among nodes. The first one focuses on balance only. The other two have a secondary goal of also improving spatial locality, by attempting to place objects on the node where they are used.
- **A heavily contended lock inside the parallel phase of Parallel Scavenge.** To avoid this bottleneck, we remove the lock: (i) by replacing a queue with a lock-free one, and (ii) by simplifying the synchronisation protocol to begin and terminate the GC's parallel phase.

The resulting GC, called NAPS, for Numa-Aware Parallel Scavenge, distributes live objects among nodes in a balanced way, and

* This research is supported in part by ANR projects Prose (ANR-09-VERS-007-02) and ConcoRDanT (ANR-10-BLAN 0208).

ASPLOS'13, March 16–20, 2013, Houston, Texas, USA.

[1] A NUMA (Non Uniform Memory Access) architecture is a multicore architecture where memory access latency depends on the locations of the physical memory and the core that accesses it. It consists of several *nodes*, each containing either one single core, as in the Tile-Gx processor family of Tilera, or several cores connected by a bus, as in the Magny-Cours of AMD. Nodes are connected by a network that we call the *interconnect*, on which several *memory controllers* are also connected.

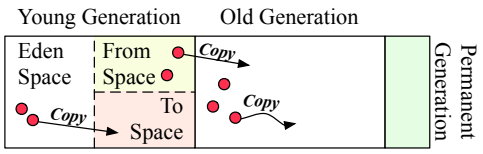

Figure 1. Memory Layout of Parallel Scavenge

completely avoids any locking during the parallel phase of the collection.

Our approach deliberately uses the simplest possible techniques, resulting in fewer than 1000 modified lines of code. This result shows that a careful adjustment of a small portion of the code is sufficient to solve a known problem, for which a more drastic redesign is often proposed.

Our evaluation is based on the applications from three industry-standard benchmark suites, SPECjbb2005 [25], SPECjvm2008 [26] and DaCapo-9.12 [4], on a 48-core AMD Magny-Cours machine with eight nodes. Our results show that:

- NAPS scales well with the number of cores. At constant heap size, the collection time of NAPS decreases with the number of cores, for applications with high memory requirements. As compared to Parallel Scavenge, NAPS never degrades performance.
- As a side-effect of balancing the memory access among nodes, in addition to the improved collection time, NAPS improves application time by up to 28%. The combination of GC-time improvement and better application performance doubles the throughput of SPECjbb2005, the most memory-intensive application in our evaluation.
- We observe that the major improvement comes from balancing memory access among nodes, which eliminates the top bottleneck to scaling GC. Beyond 32 cores, improving memory locality also becomes important, in order to decrease the load on the interconnect. In contrast, thanks to improved memory locality, the effect of decreasing memory access latency, is marginal.

The rest of the paper is organised as follow. Section 2 describes the design of Parallel Scavenge. Section 3 describes its bottlenecks and how we solve them. Section 4 reports the evaluation results and discusses their implications. Section 5 presents an overview of related work, and Section 6 concludes the paper.

2. Parallel Scavenge

This section describes Parallel Scavenge, the baseline of our study. Parallel Scavenge is the default GC of OpenJDK's HotSpot virtual machine and is the one that provides the best performance when the number of cores increases [10]. It is a *stop-the-world* collector. To achieve the best performance, it combines several advanced GC techniques: generational, copying and compacting [12]. A major advantage of the stop-the-world design is that integrating these advanced techniques is straightforward.

2.1 Overview

Empirical studies across a wide range of applications and languages show that "most objects die young:" a recently-allocated object has a greater probability of becoming garbage than one that already survived GCs [2, 13, 31]. A generational collector exploits this by segregating the heap into multiple generations, and by collecting the young generation more frequently than the older one(s).

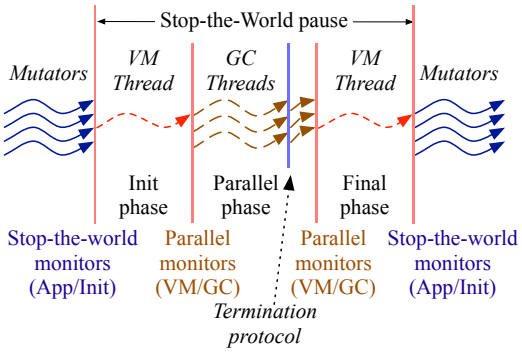

Figure 2. Phases of Parallel Scavenge

As presented in Figure 1, Parallel Scavenge defines three generations: a small *young* generation; a larger *old* generation; and a *permanent* generation, similar to the old one, but much smaller.

The young generation of Parallel Scavenge is divided into three spaces, an *eden space* and two *survivor spaces*, called *from-space* and *to-space*. When a mutator, i.e., an application thread, allocates an object, it resides initially in the eden space. If an object survives the first collection, the GC copies it to one of the survivor spaces, and, if it survives the second collection, the GC copies it to the old generation. Thus, an object must survive a complete cycle between two collections before being promoted to the old generation.

The old and the permanent generations consist each of a single space. The old space contains objects that were promoted from the young generation, and the permanent space the Java classes only.

2.2 Allocation

To avoid synchronisation between the mutators at each allocation, a mutator first allocates, for its exclusive use, a large memory chunk from the eden space called a Thread-Local Allocation Buffer (TLAB). Thereafter, it allocates objects from its TLAB, without synchronisation, using a bump pointer. Similarly, promoting an object to the survivor space or to the old generation allocates from Promotion-Local Allocation Buffers (PLAB), which are allocated from the to-space and the old generation.

2.3 Young generation collection

A young-generation collection is triggered when an allocation fails both to find space in the current TLAB and to get a new TLAB from the eden space.

When a young-generation collection begins, the eden space contains the most-recently allocated objects, the from-space contains those that survived the previous collection, and the to-space is empty. As illustrated in Figure 1, a young-generation collection copies live objects from eden space to to-space, and from from-space to old generation. At the end of a young-generation collection, eden space and from-space contain dead objects only, and are thus considered empty. At this point, from- and to-space are swapped: the from-space of the last collection becomes the to-space for the next one, and vice-versa.

As illustrated in Figure 2, Parallel Scavenge consists of several phases. First, mutators are paused at the stop-the-world barrier. During the initialisation phase, a single thread, called the VM thread, prepares GC tasks, which are executed in the parallel phase by the GC threads. A *Termination Protocol* detects that all GC threads have terminated the parallel phase; after which a barrier suspends the GC threads and wakes up the VM thread. In the final phase, the VM thread mainly resizes the heap and wakes up the mutators.

2.3.1 Synchronisation mechanism

As synchronisation between threads is a common performance bottleneck on a multicore machine [15], we describe how it is implemented and used in Parallel Scavenge. Parallel Scavenge uses monitors for this purpose. A monitor contains both a condition variable, to suspend and wake up threads, and a lock to prevent concurrent accesses to shared data. Furthermore, since the condition variable of a monitor is a shared variable, it is protected by the associated lock.

Parallel Scavenge uses two monitor pairs: *the stop-the-world pair* and the *parallel pair*. The *stop-the-world pair* suspends the mutators and wakes up the VM thread at the beginning of the collection, and vice-versa at the end. It consists of the App monitor, to synchronise the mutators, and the Init monitor, to synchronise the VM thread. The *parallel pair* suspends the VM thread and wakes up the GC thread at the beginning of the parallel phase, and vice-versa at the end. It consists of the VM monitor to synchronise the VM thread; and the GC monitor to synchronise the GC threads, and to protect data shared between the GC threads.

2.3.2 Initialisation phase

Being a stop-the-world collector, Parallel Scavenge has to ensure that all mutators are paused at the beginning of a collection. The stop-the-world monitor pair coupled with a counter to know the number of suspended mutators serves this purpose. As soon as all the mutators are suspended, the VM thread initialises the queue of GC tasks. Then, the VM thread synchronises with the GC threads: it wakes up all the GC threads that are waiting on the GC monitor, and suspends itself by waiting on the VM monitor.

2.3.3 Parallel phase

A GC thread fetches tasks from the global queue of GC tasks using the GC monitor's lock of the parallel pair to avoid concurrent fetches. In the case of young-generation collection, there are three kinds of tasks: root tasks, steal tasks and a single final task. Root tasks contain the entry points of the object graph. Steal tasks are used to balance the load between the GC threads. They are ordered in the queue after the root tasks, and are fetched by GC threads once they have fetched all the root tasks. The final task is used to elect a leader to coordinate the end of the parallel phase.

Root tasks. A root task provides an entry point into the graph of live objects. This includes static variables, mutator stacks, and inter-generation references. A GC thread performs a breadth-first traversal (BFT) of the graph of live objects, starting from the addresses provided by the root tasks.

For each object that a GC thread reaches, it creates a copy in the PLAB appropriate for the object's age, and installs a forwarding pointer to the new object in the old object's header. To avoid copying the same object concurrently with another thread, the GC thread uses an atomic compare-and-swap (CAS) instruction to install the forwarding pointer. Then, the GC thread pushes all the references contained in the object into a local *BFT queue*, which is implemented as a lock-free bounded queue, backed by an overflow stack. After this, the GC thread repeatedly pops a reference from its BFT queue, which it processes in the same way. When its BFT queue is empty, the GC thread goes back to the global task queue, popping either another root task, or eventually a steal task.

Steal tasks. As the sub-graph reachable from a root task depends on mutator activity, the load of the GC threads could be unbalanced, resulting in some GC threads remaining idle, while others still have a large amount of work. To better balance the load, an idle GC thread "steals" a reference from a randomly-chosen BFT queue and processes the corresponding sub-graph.[2] To avoid conflicting concurrent accesses to a BFT queue, the queue owner pushes and pops from the queue head, whereas other GC threads pop (steal) from the tail.

A GC thread may be unable to steal, either because the parallel graph traversal is terminated, or because its random choice never picked an overloaded GC thread even though one exists. To handle the second case, the GC thread does not directly leave the steal task. Instead, after a number of unsuccessful steal attempts, it enters a *termination protocol*. It first atomically increments a global thread counter in order to indicate to other GC threads that it is in the termination protocol. Then, the GC thread actively polls the global thread counter in a bounded loop. If the counter reaches the number of GC threads, this means that all BFT queues are empty and therefore that the parallel phase is terminated. In this case, the GC thread leaves the termination protocol. Otherwise, if the counter has not reached the number of GC threads in the bounded loop, the GC thread "peeks" into the other BFT queues. If all queues are empty, the GC thread also leaves the termination protocol, otherwise, it atomically decrements the global thread counter and continues to steal.

Final task. The final task aims to ensure that no GC thread is modifying the heap when the VM thread restarts. Once a GC thread has left the termination protocol, it again attempts to pop a task from the task queue. Since there is a single final task, only one GC thread succeeds. It thereby becomes the leader, and coordinates the other GC threads using the parallel monitors (VM and GC). We call it the final thread.

The other GC threads find the task queue empty. They increment a global thread counter protected by the GC monitor. Then, they wake up the final thread and suspend themselves on the same monitor. Conversely, the final thread waits on the GC monitor, until the global thread counter reaches the number of GC threads. Once this is done, the final thread wakes up the VM thread using the VM monitor. This constitutes entry into the final phase, where the VM thread is the only one running.

Note that, in the normal case studied here, this final synchronisation is redundant with the termination protocol because both ensure that all the GC threads have terminated their steal tasks. It is required only in specific configurations (out of scope here) where there are no steal tasks.

2.3.4 Final synchronisation phase

The main purpose of the final phase is to adapt the sizes of the spaces and the generations. The sophisticated resizing policy of Parallel Scavenge is essential for performance, because it adapts the heap size to the needs of the application, based on factors such as space usage and/or time taken by the collection cycle. In Parallel Scavenge, resizing is cheap, as it consists of simply adjusting a set of pointers.

Once this is done, the VM thread resumes the mutators using the stop-the-world monitor pair. It wakes up the mutators with the App monitor, and then sleeps, awaiting the next collection with the Init monitor.

2.4 Old-generation collection

If an object promotion fails at any stage of a young-generation collection because the old space is full, this indicates that the old generation needs to be collected. The GC thread that suffers the promotion failure sets a flag and continues the parallel phase. After the parallel phase, if the flag is set, the VM thread starts an old-generation collection. As Parallel Scavenge directly allocates in the

[2] Stealing is done from the queue only. The overflow stack is accessed solely by the owning thread.

old generation large objects that do not fit in the young one, an old-generation collection can also start during an allocation from the mutator.

Old-generation collection uses a two-phase mark-compact algorithm. In the first phase, the GC threads mark in parallel the live objects starting from the roots. In the second phase, the GC threads compact in parallel live objects by sliding them into holes created by dead objects.

Before each phase, the VM thread initialises the task queue with appropriate tasks, then starts the parallel phase. The parallel phase terminates using the same mechanism as the young generation: with the termination protocol and with the final task. They are executed twice, once for the marking and once for the compacting phase.

3. Bottlenecks and optimisations

We now analyse some of the bottlenecks experienced by Parallel Scavenge on large multicores. For each one, we describe the solution that we implement. This information is summarised in Table 1. Our experimental evaluation, presented in Section 4, shows that our optimisations improve GC scalability substantially.

3.1 Synchronisation Primitives

Parallel Scavenge uses locks to synchronise access to internal shared data structures, such as the GC task queue and the condition variables. To implement lock acquisition, Parallel Scavenge first attempts a fast-path atomic compare-and-swap (CAS) instruction, spinning for some number of iterations. If this fails, the mutator falls back to a slow path using Posix synchronisation primitives. CAS works fine for a small number of threads, as the fast path generally succeeds. However, as observed by Lozi et al. [15], on a large multicore, lock performance collapses under contention; this is particularly severe when spinning on CAS. To avoid this issue, we study the code to find the most contended locks, which we fix as explained next.

3.1.1 Lock-free GC task queue

Issue. Parallel Scavenge synchronises access to the task queue by using the GC monitor's lock. At the beginning of the parallel phase, all the GC threads access the task queue at the same time. The lock becomes contended, and its performance degrades drastically. Many GC threads wait for a long duration, preventing them from participating in the parallel phase, sometimes for the whole duration of a collection.

Solution. The task queue has First-In-First-Out semantics. In Parallel Scavenge, it is implemented as a singly-linked list. To avoid the performance collapse caused by lock acquisition, we implement the task queue with a Michael-Scott lock-free queue [18]. Recall that the VM thread enqueues tasks during the initialisation phase, which are dequeued during the parallel phase. Thus, there is no concurrency between enqueue and dequeue operations, but only between dequeues. Therefore, the VM thread enqueues without synchronisation, and GC threads dequeue using atomic CAS operations.

3.1.2 Lazy GC parking

Issue. When a GC thread executes the final task, all GC threads request the GC monitor's lock in order to synchronise the end of the parallel phase. However, this lock request immediately follows the termination protocol, and therefore all the GC threads reach this point at roughly the same time. As a consequence, the lock gets contended and its performance collapses.

Solution. To avoid this issue, we remove the GC monitor's lock. The lock was used for three purposes: first, to protect the task queue; second, to protect the global thread counter which is used for the barrier at the end of the parallel phase; and third, to protect the associated condition variable of the GC monitor, which is used to suspend the GC threads.

The first use is already taken care by the lock-free task queue. For the second case, we remove the redundant synchronisation in the final task (see Section 2.3.3). Instead of waiting for the other GC threads, the final thread simply wakes up the VM thread, and then suspends itself. Other GC threads suspend themselves without any synchronisation. After this change, the global thread counter is not required anymore.

For the last case, we replace the condition variables of VM and GC monitors with Linux's `futex_wait` calls [9]. A `futex_wait` has the semantics of an atomic compare-and-sleep, and does not require acquiring a lock.

However, our modifications potentially introduce a new race condition if a GC thread gets preempted after the termination protocol, but before suspending itself. In this scenario, the VM thread may wake up the mutator threads, a new collection may begin and the VM thread may wake up the GC threads to begin the new parallel phase. If the delayed GC thread suspends itself on the futex at this step, it will never be woken up by the VM thread, leading to a deadlock.

This is not a problem in Parallel Scavenge because, during the whole execution of the final task, the GC threads own the lock of the GC monitor. Acquiring this lock is required to wake up the GC threads with the GC monitor to begin the new parallel phase. Therefore, the GC threads inevitably suspend themselves before receiving the wake up notification. However, this condition does not hold in NAPS, because the GC monitor does not exist anymore.

To solve this problem, we use a timestamp, which is incremented atomically by the VM thread before the parallel phase. To suspend itself, a GC thread uses the futex to atomically check whether the timestamp has been modified before sleeping. If it was not modified, the futex call suspends the GC thread. Otherwise, it returns, and the GC thread directly enters the new parallel phase, thus avoiding the deadlock.

3.2 NUMA heap

When the garbage collector scans an object, this implies that it was not scanned previously in the same garbage collection cycle. Thus, GC does not benefit from hardware data caches, and accesses physical memory mostly, stressing the memory controllers. GC also stresses the interconnect between the nodes for remote accesses. These two hardware components saturate if their load is too high. In this section, we study the factors that impact performance of memory access, and consider three alternative heap layouts aiming to optimise memory placement.[3]

3.2.1 Influence of memory placement

In order to propose efficient memory layout for NUMA hardware, we first evaluate the scalability impact of four different memory placements on our 48-core/8-node AMD machine. This experiment measures the impact of memory access imbalance and locality, the main factors that impact performance [5].

Figure 3 summarises this evaluation. It contains one curve per placement algorithm, plotting the speedup in completion time of a fixed number of operations executed by a varying number of threads. Each operation consists of sequentially reading from a thread-local buffer that does not fit in the CPU caches, triggering a physical memory access. Threads are pinned on the nodes with a round-robin policy. The curves are as follows:

[3] Similar techniques already existed in Parallel Scavenge, but their implementation was flawed, as discussed.

Bottleneck	Optimisation name	Optimisation description
Heavily contended GC lock (3.1)	Lock-free GC task queue	GC task queue lock-free
	Lazy GC parking	GC thread sleeps lazily at end of parallel phase
NUMA heap (3.2)	Interleaved space	Balances the memory access between the nodes
	Fragmented space	Balance the memory accesses and improve the spatial locality
	Segregated space	Node-local collection

Table 1. Summary of proposed optimisations

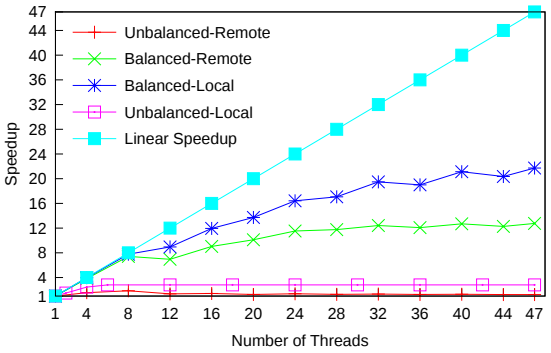

Figure 3. Influence of the memory placement.

- **Unbalanced-Remote.** All the buffers are allocated on Node 0. Access is (i) remote (except for the threads running on Node 0), and (ii) unbalanced.
- **Balanced-Remote:** The pages of the buffer are allocated to the nodes in round robin. Access is (i) mostly remote, and (ii) perfectly balanced.
- **Balanced-Local:** The buffers are allocated on the same node as its corresponding thread. Access is (i) local only, and (ii) perfectly balanced.
- **Unbalanced-Local.** The threads are first pinned on Node 0, and only these ones participate. The curve is extended beyond Node 0 by adding idle threads. Access is (i) local only, and (ii) unbalanced.

As shown in Figure 3, the performance of *Unbalanced-Remote* improves until 8 threads only, whereas *Balanced-Remote* continues to improve up to 32 threads. We conclude that when load is unbalanced, fewer threads suffice to saturate the machine (either the memory controllers and/or the interconnect between the nodes). This experiment shows that balancing the load has a huge impact on scalability. Comparing *Unbalanced-Remote* with *Unbalanced-Local* shows that balancing load is a higher priority than improving locality.

Furthermore, *Balanced-Remote* stops improving beyond 32 threads, whereas *Balanced-Local* continues to improve up to 40 threads, and has significantly better results. We conclude that improving memory locality constitutes a secondary scalability objective, once balancing access among nodes has been taken care of.

3.2.2 Memory access imbalance

Issue. The memory access imbalance issue mainly concerns eden space, because of the conjunction of two things: the kernel's memory allocation policy; and a sequential pattern of memory initialisation by the application, as we explain next.

Parallel Scavenge reserves a large virtual memory range for the heap using the `mmap` system call. No physical pages are initially allocated to an `mmap`ped region; the first time a virtual page is accessed, the kernel allocates a physical page on demand. The kernel's allocation policy preferentially allocates a physical page from the same node that triggered the fault [14]. Thereafter, a virtual address range remains associated to the same physical node.

Many applications start with a single-threaded initialisation phase, which consumes most of the eden space. Because of the kernel allocation policy, these objects are all allocated on the same physical node, the one executing the initialisation thread. The physical pages mapped to the eden space remain on that same node thereafter, causing this node to suffer memory access imbalance. This allocation pattern is typical of the *Unbalanced-Remote* case discussed earlier.

This issue can also occur with the other spaces when threads create large objects. Parallel Scavenge allocates these objects directly in the old space, also leading to the mapping of large part of the virtual space to a single node.

Interleaved spaces. To address this issue, we introduce interleaved spaces, whose pages are mapped from different nodes with a round robin policy. By balancing memory allocation among the nodes, we expect that interleaved spaces will also balance memory access. Our experiments in Section 4 support this hypothesis.

3.2.3 Memory locality and access balance

Issue. Interleaved spaces balance the load but do not address memory locality. However, as shown by comparing the curves *Balanced-Remote* and *Balanced-Local* in Figure 3, excessive remote access can be a scalability bottleneck. Therefore, we introduce two heap layouts that balance load and secondarily improve memory locality.

Fragmented spaces. Fragmenting consists of dividing a space into multiple fragments, where each fragment is a virtual address range that gets all its physical pages from a single node. Under this policy, a thread allocates an entire TLAB or PLAB from the fragment associated with the node where it is executing.

Fragmented spaces improve locality for the mutators because: (i) a thread mostly accesses objects that it allocated recently, and (ii) the operating system avoids migrating threads between nodes. It also helps to improve locality for the GC, because objects are copied to the memory node of the GC thread that performs the copy.

Fragmented spaces also solve the issue of memory access imbalance caused by the initialisation phase of the application, because the page allocation policy of each fragment is fixed, and thus insensitive to the application's allocation pattern. Additionally, in conjunction with the work-stealing mechanism, fragmented spaces help to balance memory accesses among the nodes. Work-stealing ensures that every GC thread copies roughly the same number of bytes; hence it eventually balances the distribution of objects among nodes, even if the objects were initially allocated on a single node (we pin GC threads, spreading them across all the nodes in round robin).

However, fragmented spaces do not help to balance memory access in the specific case where the application allocates objects according to a master-slave pattern, in which a master thread allocates the objects for the slaves. With a fragmented space, objects

Space type	Load balancing	Mutator locality			GC locality				
		First Access	Later Access New	Later Access Promoted	Eden Scan	Survivor Copy	Survivor Scan	Old/Permanent Copy	Old/Permanent Scan
Parallel Scavenge	−	+	−	−	−	−	−	−	−
Interleaved space	+	−	−	−	−	−	−	−	−
Fragmented space	∼	+	+	−	−	+	−	Not Applicable‡	
Segregated space	∼	+	+	+	+	+	+	Not Applicable‡	

‡ Our simple algorithms are not adequate for compacting spaces.

Table 2. Load balancing vs. locality properties

will all be allocated on the master node, leading to an unbalance of the accesses. After a collection, the balance will be re-establish, but, as threads mostly access recently-allocated objects, interleaved spaces are better. The only of the applications that we evaluated to exhibit this pattern is H2.

Segregated spaces. A segregated space is a fragmented space that is restricted to being accessed by GC threads running on the same node. This is achieved by: (i) restricting work-stealing to GC threads that run on the same node; and (ii) by disallowing a GC thread from accessing a remote object. When a GC thread discovers a reference to an object located on some other node, rather than scanning it itself, it sends a "scan" message to the GC threads running on remote node.

Segregated spaces ensure perfect memory locality, at the cost of exchanging messages between nodes. Moreover, segregated spaces solve the issue of memory access imbalance caused by the the initialisation phase of the application, using the same approach as described earlier for fragmented spaces. However, since segregated spaces never move an object between nodes, if the allocation rate of different mutator threads is asymmetric, object placement will remain unbalanced.

3.2.4 Summary of the space policies

Our analysis of load-balancing vs. locality trade-offs is summarised in Table 2. For a given space type, the second column identifies whether the approach is systematically good (+) or bad (−) for load balancing, or if we observe that it is good (∼) due to the memory allocation pattern of the applications. The other columns depict the locality properties, for the mutator and the GC respectively. For the mutator, we distinguish the mutator's initialisation phase (First) from the rest of the execution (accesses to new and promoted objects). For GC, we distinguish the different generations, and within each generation, between scanning and copying the object.

Mutator locality is improved by locating newly-allocated objects on the mutator's execution node (First Access, Later access/ New) and by copying objects to their node of use (Later Access/ Promoted). GC locality is improved by scanning objects located in the node where the GC thread runs (Scan), and by copying objects towards the node where the GC thread runs (Copy).

3.3 Implementation details

Parallel Scavenge already provides interleaved and fragmented spaces through a command line option. However, this implementation is inefficient on our hardware. To provide evidence of the problem, we evaluate a 4.5 min.-run of SPECjbb2005, our most memory-intensive application, with 48 GC threads and 48 mutators. In this test, activating the fragmented-space option slows Parallel Scavenge down even more (63.6 GB/s with fragmented spaces, vs. 98.4 GB/s in the default configuration, a slowdown of 35%). We explain the issues hereafter.

The resizing problem. The fragmented spaces of Parallel Scavenge raise two different issues. First, in the Parallel Scavenge fragmented spaces, each fragment is $1/N^{\text{th}}$ of the size of the space (where N is the number of nodes). Therefore, an allocation might fail prematurely when a single fragment is full, even though there is still space in other fragment(s). This totally invalidates the decisions of the resizing policy, and can lead to many useless collections. In our SPECjbb test, we observe that the number of collections triggered with fragmented spaces is almost twice the number without (325 collections versus 177).

Second, resizing a space is substantially more expensive. As the spaces and generations are contiguous in virtual memory, resizing requires many remappings. Each remapping triggers two system calls, one to free the physical pages from their virtual addresses, and another one to re-associate the virtual address range to a different memory node. This cost increases with the number of remapped pages, i.e., with the size of the fragmented space. In our SPECjbb test, we observe that roughly 20% of collection time is spent in resizing.

Solution. Our own implementation of fragmented spaces leverages the difference between virtual and physical memory. We initially reserve a very large virtual space for the heap, and use allocation counters to control the number of physical bytes actually allocated in a space. This large virtual memory reservation is possible on a 64-bit machine because the virtual address space is sufficiently large.

We avoid invalidating the resizing policy by making the virtual memory size of each fragment equal to the total size of the space it belongs to. This ensures that allocation from a space does not fail until its allocation counter reaches the assigned limit. Thus, NAPS triggers collection at exactly the heap size specified by the resizing policy, and does not invalidate its decisions.

We avoid the cost of system calls by reserving the maximal possible heap size. Each space and each fragment have a fixed address range, and their pages are mapped according to the memory policy configured during the initialisation of the virtual machine. As we pre-reserve the maximal size for each space, and therefore for each fragment, spaces and fragments can grow or shrink safely, without any remapping.

4. Evaluation

This section analyses the effect of our optimisations on GC pause time and on application throughput, using representative applications from standard benchmark suites (SPECjbb2005, SPECjvm-2008 and DaCapo).

4.1 Experimental Setup

4.1.1 Hardware and Operating System

We conduct our experiments on a machine with four AMD Opteron 6172 sockets, each consisting of two nodes. Each node consists of six cores (2.1 GHz clock rate). The nodes are connected to each other by HyperTransport links, with a maximum distance of two hops. In total there are 48 cores, carrying 96 GB of RAM consisting of 8 memory nodes. The system runs a Linux 3.0.0 64-bit kernel.

Application	Heap Size (GB)	Description
SPECjbb2005	3.5	Business logic service application
XML Transform	1	Transforms an XML tree
XML Validation	2	Validates an XML tree
Compiler.Sunflow	2	Compiles Sunflow Java code
Crypto AES	1	Runs AES algorithm
Eclipse	0.5	Java editor
Tradesoap	0.5	Business logic service application

Table 3. Applications used in the evaluation

		Baseline	Inter.	Fragm.	Segr.
Non-Locity	Eden	≥0.7	≥0.7	≥0.7	**0.0 0.0 0.0**
	From				**0.0 0.0 0.0**
	To			**0.0 0.0 0.0**	**0.0 0.0 0.0**
	Old			≥0.7	≥0.7
Imbalance	Eden	**2.6 2.6 1.9**	≤0.3	0.1 0.6 0.5	0.1 0.6 0.64
	From	≤0.3		≤0.3	0.2 0.2 0.4
	To				≤0.3
	Old				

Table 4. Non-locality (remote access ratio) and imbalance (standard deviation of remote access). Left to right in each column: SpecJBB2005, XML Transform and TradeSoap.

4.1.2 Benchmarks

We base our evaluation on the applications from three benchmark suites. As these applications use much less memory than our available memory, each experiment sets a heap size that is very close to the application's working set size in order to focus on the garbage collector. As we measure the throughput of the garbage collector, i.e., the number of bytes collected per second, the size of the heap has only a small impact on our result.

Table 3 summarises the applications we evaluate. We discard the less memory-intensive applications of the suites, because they are impacted only marginally by GC performance.

SPECjbb2005 is a business logic service-side application. It emulates a three-tier client/server system, with emphasis on the middle tier [25]. This benchmark runs for a fixed amount of time (configured by the user) and measures application throughput, i.e., the number of operations per second.

The benchmark allocates a warehouse for each worker thread. We run a warm-up round of 30 s with 40 warehouses, then a final round of four minutes with 48 warehouses. We use a heap size of 3.5 GB to execute this application. This is the benchmark with the largest working set, and therefore the one that stresses GC the most.

SPECjvm2008 is a suite of real-life and area-focused applications [26]. Depending on benchmark settings, an iteration runs either for some number of operations per mutator thread and measures application time, or for some amount of time and measures throughput. We use the first setting. We discard scientific applications as they do not generate sufficient work to trigger the GC more than once or twice. Therefore, from this suite, we only evalulate XML Transform, XML Validation, Crypto AES and Compiler.Sunflow. We use a heap of 1 GB for XML Transform and Crypto AES, and 2 GB for the other two.

DaCapo is a set of real-life applications, aimed at evaluating the performance of different components of a Java Virtual Machine [4]. Each application in this suite contains a predefined, fixed-sized workload. Because of their small working sets (between 100 and 500MB), the DaCapo applications do not stress the GC: at 48 cores, each GC thread has only a few kilo-bytes to collect. Still, we present the results of two representative applications from this suite, Eclipse and Tradesoap, to study the effects of our optimisations on less memory-intensive applications. They both require a 500 MB heap.

4.2 Analysis of individual optimisations

This section discusses the performance impact of each optimisation. We focus on three applications for this analysis: SPECjbb-2005, XML Transform from SPECjvm, and Tradesoap from DaCapo. We choose these three because they represent very different working sets, from small for Tradesoap to large for SPECjbb-2005. For XML Transform, we run two iterations, each consisting of 40 operations per mutator; for Tradesoap, we run 10 iterations with *large* workload size. We execute each experiment three times and report the average. The standard deviation between the three runs is not significant.

In order to focus on the performance of the garbage collector as the number of cores increases, we keep the memory allocated constant by setting the number of application threads to 48. We measure GC scalability by varying the number of GC threads from one to 48, in steps of eight.

- **Table 4** evaluates, for the three applications (left to right: SpecJBB2005, XML Transform and TradeSoap), memory access non-locality and memory access imbalance. Non-locality is defined as the fraction of memory accesses during collection that are remote. For example, 0.7 means that 70% of GC accesses are remote, whereas 0.0 means that all GC access is local. Imbalance is measured as the standard deviation over the ratio of number of accesses to a node divided by the total number of accesses. For example, an imbalance of 0.1 means that the standard deviation is small and thus the load is balanced, whereas an imbalance of 2.6 reports a large standard deviation, and thus the load is highly unbalanced.
- **Figure 4** plots GC throughput, i.e., the total amount of memory allocated by the application, divided by the total amount of time spent in GC. The throughput characterises the pause time of the applications independently from the number of allocated bytes. This shows the effects of our optimisations on GC performance.
- **Figure 5** plots completion time of Tradesoap and XML Transform (lower is better) and throughput for SPECjbb (higher is better). This shows the effects of our optimisations on the GC and on the application in combination.
- **Figure 6** highlights the effect of our optimisations on the application, in addition to the GC effect, by plotting the difference between completion time and GC pause time. We report curves for XML Transform only. We discard SPECjbb because it is not possible to isolate application improvement from GC improvement with a fixed duration run. We discard Tradesoap as it uses very little memory to show any significant results.

4.2.1 Interleaved spaces

Impact on garbage collection. Recall, from Section 3, that many applications initialise using a single thread, which maps the physical pages of the eden space to a single node. For example, with SPECjbb2005, at the time of first collection, we observe that 95% of the eden space's pages were allocated from a single node. As shown in Table 4, all three applications suffer from this problem, and memory access is highly unbalanced between the nodes. Interleaved spaces solve this problem by completely avoiding load imbalance.

As shown in Figure 4.a, we observe that with 48 threads, interleaving improves GC throughput for SpecJBB2005 approximately by 75% as compared to Parallel Scavenge. XML Trans-

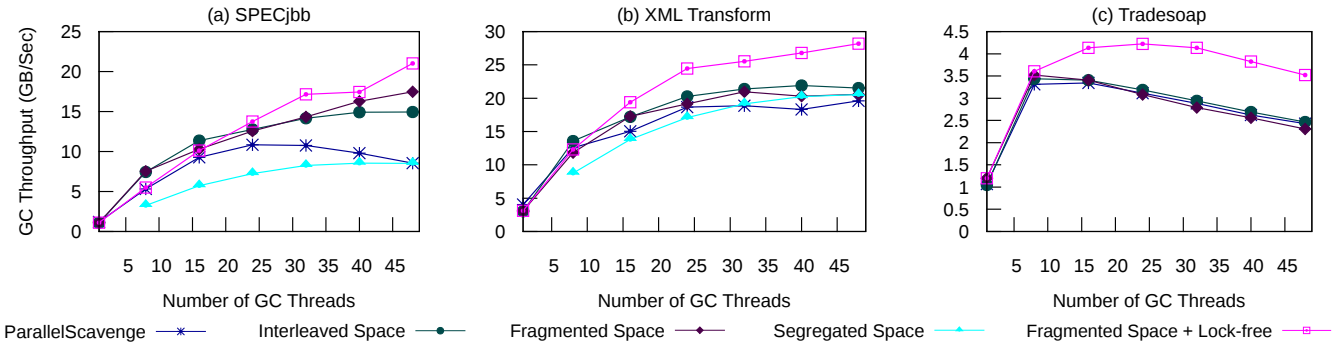

Figure 4. Impact of each of the optimisations on GC throughput

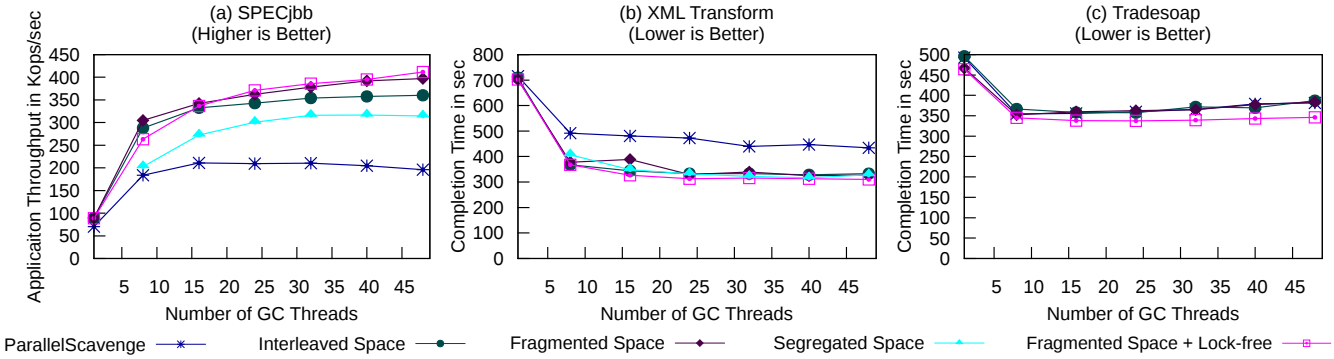

Figure 5. Impact of each of the optimisations on application performance

form demonstrates exactly the same situation, with GC throughput increasing by roughly 10% at 48 cores (Figure 4.b). Since this application allocates less memory, the GC spends less time in graph traversal, and therefore, the effect of interleaved spaces is less significant. In Tradesoap, we observe that even though pages are mainly allocated from two nodes in the beginning, and that the application suffers from a high imbalance (see Table 4), using an interleaved space only leads to a 1% improvement (Figure 4.c). This is because the heap is very small and graph traversal is dominated by the time spent in synchronising the threads.

We thus conclude that balancing memory accesses between the nodes has a high impact on the garbage collector for memory intensive applications and that the impact is less if the GC threads don't have enough memory to collect.

Impact on application. Better balancing memory access among nodes also impacts the application. Figures 5 and 6 show this phenomenon. For example, after excluding GC time, XML Transform runs 26% faster when using interleaved spaces. This shows that both the application and the GC are hampered by an imbalance of the memory accesses between the nodes. Overall, the results show that load balancing has a high impact on both application and GC, especially if the application is memory-intensive, such as SPECjbb.

4.2.2 Fragmented and segregated spaces

In our next experiment, we use either fragmented or segregated space for all the spaces of the young generation. However, we still use interleaved spaces for the old and permanent generations, as these generations use a compacting algorithm. Zhou and Demsky [32] propose a NUMA-aware compaction algorithm, but this is out of our scope. Furthermore, our results show that using a fragmented

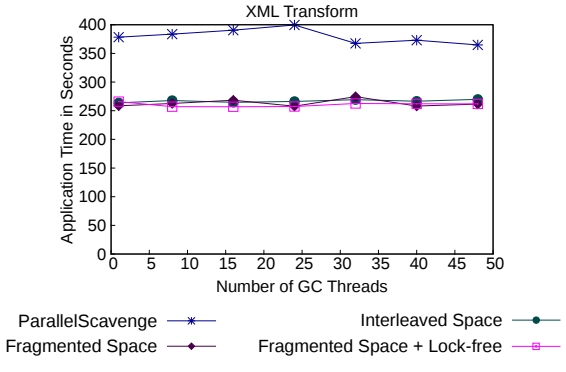

Figure 6. Application improvement not including GC.

space for the other generations is not required to make the garbage collector scale.

Fragmented spaces. Table 4 shows that fragmented spaces reduce the memory access imbalance as compared to Parallel Scavenge. For this reason, except for SPECjbb with more than 32 cores, fragmented and interleaved spaces give roughly the same improvement as compared to Parallel Scavenge (see Figure 4). For the same reason, using fragmented space leads to the same application improvement (see Figure 6).

Fragmented spaces improve locality by eliminating remote access when objects are copied to the to-space (see Table 4). They also improve locality of the application since instead of allocating objects on random nodes, as in the case of interleaved space, an object is allocated on the node that triggers the allocation. Despite this

locality improvement, Figures 4, 6 and 5 show that the performance of the GC and of the application is almost the same with either an interleaved space or a fragmented space, when the GC uses less than 32 threads. This result shows that remote access latency has only a small impact on the GC and on the application.

However, in SPECjbb, fragmented spaces improve GC performance with more than 32 GC threads, as compared to interleaved spaces (Figure 4.a). As presented in Section 3.2, good memory locality helps scalability when the load is balanced. This property explains why SPECjbb is unable to scale beyond 32 cores with interleaved spaces, whereas it continues to scale with fragmented spaces. This phenomenon is not visible in XML Transform and Tradesoap, as their heap sizes are smaller and thus, the number of memory accesses is smaller during the collection.

To summarise, we conclude that memory latency due to remote access has little impact on the GC and the application. However, increasing locality is necessary to let the GC scale.

Segregated spaces Segregated spaces require at least one thread on each node to process the received messages, and is thus plotted with at least 8 threads. Furthermore, we cannot use segregated spaces in Tradesoap, as Tradesoap uses a lot of weak references and our segregated space implementation is unable to process them in parallel.

Table 4 shows that segregated spaces eliminate remote access when objects are scanned or copied in the young generation. The table also depicts that segregated spaces are as beneficial as fragmented spaces in reducing memory imbalance. These properties explain that, as shown in Figure 4.a and 4.b, segregated spaces scale up to 48 cores.

However, segregated spaces do not perform as well as fragmented spaces. We observe on SPECjbb that 12% of the scanned objects do not belong to the node where they are discovered during scanning. Moreover, for each remote object, the scanning thread sends the object reference to a remote queue, causing an inter-node cache miss. Thereafter, a thread on the remote node, updates the location of the reference in the scanning thread's node, after copying the object; this causes another inter-node cache miss. Therefore, we believe that this extra overhead compensates any benefit due to better locality, and hence hampers the performance. We did not investigate it further, which would be an interesting extension to this work.

4.2.3 Lock-free task-queue and lazy thread parking

Individual impact. The lock-free task-queue decreases the synchronisation cost at the beginning of the parallel phase, and lazy thread parking at the end. As synchronisation overhead is most noticeable when the heap is small, this is where we expect the effect of this improvement to be most visible.

Unexpectedly, better synchronisation has an important effect on all the three applications (see Figure 4). As compared to fragmented spaces, our synchronisation optimisations improve the performance of SPECjbb, XML Transform and Tradesoap by 23%, 40% and 34% respectively, when using 48 GC threads. For XML Transform and Tradesoap, this improvement is large mainly because the time to traverse the object graph is small. For SPECjbb, the most memory intensive application, optimising the synchronisation also begins to have a significant impact after 24 GC threads and is required to make the GC scale up to the 48 cores. Indeed, without it, beyond 24 GC threads, the bouncing of the lock's cache line between the cores of the GC threads becomes significant and hampers performance by saturating the interconnect [15]. The lightweight synchronisation protocol avoids this overload.

Combined impact. Combined with the fragmented heap, the synchronisation optimisations let SPECjbb and XML Transform scale up to 48 GC threads. It is not the case for Tradesoap as its heap is small and adding more GC threads only increases contention on the cache lines that contain the objects.

	Parallel Scavenge	NAPS
SPECjbb	105 ms	49 ms
Compiler.Sunflow	108 ms	80 ms
XML Transform	14 ms	9.2 ms
XML Validation	80 ms	55 ms
Crypto AES	25.6 ms	9 ms
Eclipse	17.8 ms	13 ms

Table 5. Individual pause time

As compared to Parallel Scavenge, the pause time reduction due to fragmented spaces combined with lock optimisations translates to a total application time improvement of 9% in Tradesoap and 28% in XML Transform, and improved throughput of SPECjbb-2005 by more than twice at 48 GC threads (see Figure 5).

4.3 Scalability

NAPS uses fragmented spaces in the young generation and interleaved spaces in the others, as this combination provides the best performance. Figure 7 compares throughput of Parallel Scavenge and NAPS, i.e., total amount of memory allocated during the whole execution divided by total pause time. It would be unfair to compare throughput across applications, since GC performance depends not only on the number of allocated bytes, but also on the number of objects that survives their collection.

Observation 1: NAPS improves performance and scalability over Parallel Scavenge in all cases, except Compiler.Sunflow where both collectors roughly provide the same performance. This shows that the bottlenecks that we have identified are real.

Observation 2: The performance of NAPS continues to increase up to 48 cores with the 5 memory-intensive applications, except Crypto AES. Furthermore, by adding more GC threads, the performance of NAPS never degrades, except for Eclipse which is, along with Tradesoap, the least memory-intensive applications of the evaluated applications. For Crypto AES, NAPS stops scaling after 24 threads, and for Eclipse it degrades after 32 GC threads. For these applications, the number of objects that survive their collection is small, either because the application mostly allocates objects that die quickly (Crypto AES), or because the application requires a small heap (Eclipse).

Observation 3: The Table 5 reports, for each of the applications, the pause time of a single collection pause. As we can see, NAPS reduces the individual pause times, by up to 2.8 times in the best case. These results confirm that using a scalable stop-the-world collector improves the responsiveness of the application.

Conclusion: From these three observations, we conclude that a stop-the-world GC, such as NAPS, is able to scale almost linearly up to 48 GC threads, provided it has enough memory to collect. Moreover, using a scalable stop-the-world GC actually decreases the pause time and helps in improving the responsiveness of applications.

5. State of the Art

There has been much research on concurrent garbage collection, mainly targeting real-time applications [6, 17, 22, 23]. These collectors have been evaluated on eight cores and two nodes at most. The evaluation of Immix, a recent stop-the-world collector, has the same limitation [3]. This is insufficient to expose the problems that

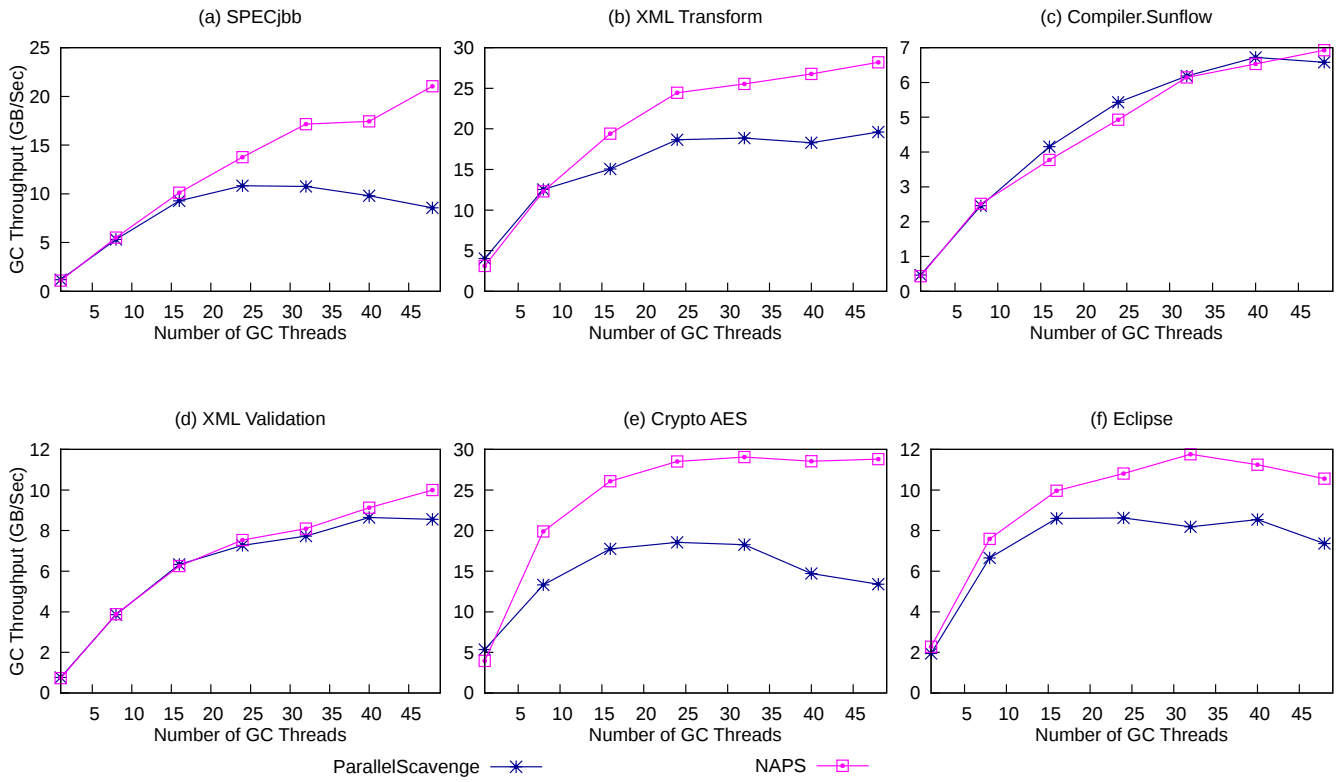

Figure 7. GC Throughput of Parallel Scavenge and NAPS.

arise in large multicores. For intance, Gidra et al. [10] show that Garbage First [6] collapses after 20 cores.

Zhou and Demsky [32] propose a new parallel mark-and-compact collector targeting the TILE-Gx microprocessor family, for which each core is a memory node [30]. It focuses on increasing memory locality, while balancing memory access across the nodes, by balancing the allocations at each node, during compaction. We do not implement such an algorithm, because we believe it will not impact performance on our hardware, in which memory-load balance is the main bottleneck, not the locality. It would be interesting to compare this algorithm with NAPS on a TILE-Gx processor, to isolate which part of the improvement is caused by improving memory locality, and which by balancing memory-load.

Ogasawara [20] and Tikir and Hollingsworth [29] ensure that an object is copied to the memory node where it is accessed most of the time. These algorithms improve application locality, but do not improve GC locality. Again, these algorithms target memory locality, not memory-load balance, the bottleneck on our hardware.

There is considerable amount of work on efficient load balancing among GC-threads. Flood et al. [8] propose a per-object work stealing technique using lock-free double ended queues, also used by Marlow et al. [17]. This is the scheme implemented in OpenJDK 7. Our experiments show that this algorithm is well-adapted to our hardware platform, since, by balancing the load across GC threads, it balances copies across nodes, and hence balances the load across the memory controllers. Oancea et al. [19] associate work-lists to dedicated partitions of the heap. Only the owner of a work-list (the worker thread that works exclusively on that work-list) traces the objects in the corresponding heap partition. For load balancing, workers take ownership of whole work-lists, rather than of individual objects. We believe that stealing a whole work-list is too coarse, and will result in unbalanced memory-load.

A widely-used technique to improve locality and concurrency is to use thread-local heaps [1, 7, 16, 24, 27]. In such GCs, an object is first allocated in a thread-local heap, and gets migrated to a shared global heap only when it gets referenced from the shared heap. This heap layout ensures that a thread-local heap collection can execute concurrently with the application threads on other cores. These algorithms do not focus on efficient collection of the shared heap on large-scale multicore hardware. Moreover, even if the read barrier optimisation of Sivaramakrishnan et al. [24] is applied, applications must still pay the price of a write barrier.

Iyengar et al. [11] states that "Stop-the-World garbage collection is a fundamental fault line in the design of managed runtime environments [because] such stop-the-world techniques become untenable for online processing application [when the heap size grows]". Our results do not support this statement; on the contrary, well-established parallel computing techniques appear to be sufficient to enable stop-the-world collectors to scale on multicore hardware.

6. Conclusion

This paper studies the problem of scalability of throughput-oriented GCs on multicore hardware. We present how we can fix these bottlenecks using well-established parallel programming techniques, and show that they were not caused by the stop-the-world design, but by some of the mechanisms that were not planned for contemporary NUMA multicores with tens of cores. Our evaluation suggests that today, there is no conceptual reason to believe that the pause time of a stop-the-world GC will increase with the increasing number of cores and memory size of multicore hardware. It leads to the conclusion that the costly and complex fine-grain synchronisation of concurrent GCs are not yet required for throughput-oriented

GCs: to achieve good performance, and to scale with the increasing number of cores and heap size.

As a future work, we will explore how we can make segregated spaces perform better by avoiding most of the inter-node messages. We propose to make the old generation fragmented, which will enable us to identify most of the references in their home nodes locally.

Acknowledgement

We would like to thanks our anonymous reviewers for their insightful comments and suggestions.

References

[1] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM '10*, pages 21–30. ACM, 2010.

[2] A. W. Appel. Simple generational garbage collection and fast allocation. *SP&E*, 19(2):171–183, 1989.

[3] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI '08*, pages 22–32. ACM, 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190. ACM, 2006.

[5] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. In *ASPLOS '13*. ACM, 2013.

[6] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM '04*, pages 37–48. ACM, 2004.

[7] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *POPL '93*, pages 113–123. ACM, 1993.

[8] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *JVM '01*, pages 21–21. USENIX Association, 2001.

[9] H. Franke and R. Russell M. K. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium, OLS '02*, pages 479–495, 2002.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. In *SOSP Workshop on Programming Languages and Operating Systems, PLOS '11*, pages 1–5. ACM, 2011.

[11] B. Iyengar, G. Tene, M. Wolf, and E. Gehringer. The collie: a wait-free compacting collector. In *ISMM '12*, pages 61–72. ACM, 2012.

[12] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 1st edition, 2011.

[13] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 26(6):419–429, 1983.

[14] LinuxMemPolicy. What is linux memory policy? http://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt, 2012.

[15] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote Core Locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC '12*, pages 65–76. USENIX Association, 2012.

[16] S. Marlow and S. Peyton Jones. Multicore garbage collection with local heaps. In *ISMM '11*, pages 21–32. ACM, 2011.

[17] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM '08*, pages 11–20. ACM, 2008.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96*, pages 267–275. ACM, 1996.

[19] C. E. Oancea, A. Mycroft, and S. M. Watt. A new approach to parallelising tracing algorithms. In *ISMM '09*, pages 10–19. ACM, 2009.

[20] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying GC. In *OOPSLA '09*, pages 377–390. ACM, 2009.

[21] OpenJDK Memory. Memory management in the Java hotspot™ virtual machine. Technical report, Sun Microsystems, 2006.

[22] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. Stopless: a real-time garbage collector for multiprocessors. In *ISMM '07*, pages 159–172. ACM, 2007.

[23] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *PLDI '10*, pages 146–159. ACM, 2010.

[24] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan. Eliminating read barriers through procrastination and cleanliness. In *ISMM '12*, pages 49–60. ACM, 2012.

[25] SPECjbb2005. SPECjbb2005 home page. http://www.spec.org/jbb2005/, 2012.

[26] SPECjvm2008. SPECjvm2008 home page. http://www.spec.org/jvm2008/, 2012.

[27] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM '00*, pages 18–24. ACM, 2000.

[28] G. Tene, B. Iyengar, and M. Wolf. C4: the continuously concurrent compacting collector. In *ISMM '11*, pages 79–88. ACM, 2011.

[29] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *IPDPS '05*, pages 108–117. IEEE Computer Society, 2005.

[30] Tilera. TILE-Gx processor family. http://www.tilera.com/products/processors/TILE-Gx_Family, 2012.

[31] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *SDE '84*, pages 157–167. ACM, 1984.

[32] J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In *ISMM '12*, pages 3–14. ACM, 2012.