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

Anonymous submission

ABSTRACT

Large-scale multicore architectures are problematic for garbage collection (GC). In particular, throughput-oriented stop-the-world algorithms demonstrate excellent performance with a small number of cores, but have been shown to degrade badly beyond approximately 20 cores on 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.

1. INTRODUCTION

Modern multicore hardware creates new challenges to the performance of garbage collection (GC) for CPU-intensive applications that require high throughput. To exploit these cores, throughput-oriented GCs are parallel and use as many threads as the number of cores to collect the memory. But, as reported by Gidra et al. [9], on a 48-core machine, the three throughput-oriented GCs used in OpenJDK 7 degrade beyond about 20 GC threads, taking over one-third of total application time at 48 GC threads. We confirmed this result on SPECjbb2005 [23] and on half of the most memory-intensive applications of SPECjvm2008 [24].

This poor performance scalability raises the question of whether the design of such throughput-oriented GCs is adequate. To achieve the best possible performance with few cores, they are stop-the-world (as opposed to concurrent): the collector pauses the application threads during the whole collection to prevent concurrent memory access from the application. This avoids the complex fine-grain synchronisation between the collector and the mutator, and the costly instrumentation of application code required to intercept reference mutations [1, 6, 10, 14, 22, 25, 26].

According to some authors, for instance Iyengar et al. [10], stop-the-world GC cannot perform well on large machines. They argue that the amount of memory that an application allocates per time unit increases proportionally to the number of cores, causing pause-time to degrade because stop-the-world GCs are unable to scale. This raises the issue weather the stop-the-world design is intrinsically inadequate for multicores, and weather they should be thrown away in favour of concurrent algorithms. This is the focus of our study: has the time come to pay the price of concurrent collectors, i.e., fine-grain synchronisation and intrusive instrumention of application code, to achieve good performance scalability?

The result of our study suggests that the answer is no, at least on an off-the-shelf 48-core NUMA machine.[1] We study Parallel Scavenge [19], the stop-the-world throughput-oriented collector used by default in OpenJDK 7. We show that its bad performance scalability 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 a single memory controller, during execution of both application and GC code. To fix this issue, we study two solutions that balance the load across all memory controllers. One aims to increase the spatial locality of data, i.e., it attempts to place objects on the node where they are used, the other not.
- A heavily contended lock inside the parallel phase of Parallel Scavenge. To avoid this bottleneck, we removed the lock: *(i)* by replacing a data structure with a lock-free one, and *(ii)* by simplifying the synchronisation protocol to start and end the GC's parallel phase.

The resulting GC, called NAPS for Numa-Aware Parallel Scavenge, distributes live objects among the nodes in a balanced way, and completely avoids any synchronisation during the parallel phase of the collector.

[1] A NUMA (Non Uniform Memory Access) architecture consists of several nodes, each containing several cores and managing a memory partition. It offers several memory controllers, either one per partition (as in the Magny-Cours, where a node contains multiple cores), or one for multiple partitions (as in the Tile-Gx processor family, where each core is a node [28]).

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 re-design is often proposed. It may be that applying more sophisticated techniques [1, 6, 14, 18, 22, 25, 27, 30] could lead to better performance, but this will be unrelated to the scalability bottleneck.

Our evaluation is based on the seven most memory-intensive applications from three industry-standard benchmark suites, one from SPECjbb2005 [23], four from SPECjvm2008 [24] and two from DaCapo-9.12 [4], on a 48-core AMD Magny-Cours machine with eight nodes. Our results show the following:

- At constant heap size, the collection time of NAPS decreases linearly with the number of cores. As compared to Parallel Scavenge, it never degrades performance.
- As a side-effect of balancing the load among memory controllers, NAPS decreases application time up to 28% (in addition to the improved collection time). The combination of GC-time improvement and better application locality doubles the throughput of SPECjbb2005.
- Improving spatial locality of the garbage collector has only marginal effect on the performance of both the garbage collector and the application. This result is due to hardware prefeteching, which pre-loads cache lines, hiding the high inter-node communication latency.

The rest of the paper is organised as follows. 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 the related works, 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. It is a Stop-the-World collector. To achieve the best performance, it combines several advanced GCs techniques: generational, copying and compacting.[2] In addition to being relatively simple, a major advantage of the stop-the-world design is that integrating these advanced techniques is straightforward.

2.1 Heap layout and basic design

Empirical studies across a wide range of applications and languages show that "objects die young:" a recently-allocated one has greater probability of becoming garbage than one that already survived GCs [2, 11, 29]. A *generational* collector exploits this by segregating the heap into generations and by collecting the *young generation* more frequently than the older one(s). As presented in Figure 1, Parallel Scavenge defines three generations: a small young generation; an *old* generation, which is comparatively much larger; and a *permanent* one, similar to the old generation, but much smaller in size, which is used to hold Java classes only.

The young generation is divided into three *spaces*, the *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. Thus, a recently-allocated object gets a second chance before moving to the old generation. Without this, an object allocated just before a collection might be aged prematurely.

[2]These terms will be defined shortly.

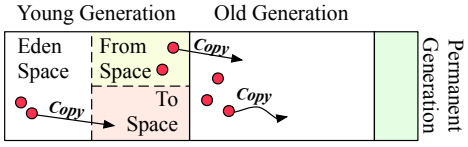


Figure 1: Memory Layout of Parallel Scavenge

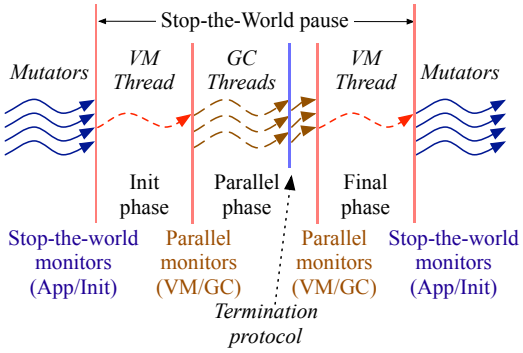


Figure 2: Phases of Parallel Scavenge

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

The old generation consists of a single space, which contains objects that were promoted from the young generation (i.e., from the from-space). The permanent generation is similar.

2.2 Allocation

To avoid synchronisation at each allocation, a mutator first allocates, for an 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 fast bump pointer. Similarly, promoting an object to survivor space or to the old generation allocates from a promotion-local allocation buffer (PLAB), which is allocated from the to-space or the old generation.

2.3 Young generation collection

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

new TLAB from eden-space.

As illustrated in Figure 2, the baseline 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 (presented in detail in the next sections), mainly a task per root set of the object graph. GC tasks are executed in the following parallel phase by several *GC threads*. In the parallel phase, a *Termination Protocol* detects that all GC threads have ended; after which a barrier suspends the GC threads and wakes up the VM thread. In the final phase, the VM thread cleans up and wakes up the mutators.

Between phases, mutators, VM and GC threads must synchronise. For this purpose, Parallel Scavenge uses monitors, which contain both a lock and a condition variable. A condition variable is used to suspend and wake up threads, and requires taking the associated lock beforehand. Parallel Scavenge defines 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.

2.3.1 Initialisation phase

Parallel Scavenge is a "stop-the-world" collector, i.e., all mutators are paused at the beginning of a collection. They resume when collection is complete. The VM thread can thus begin its work only after it has ensured that all VM threads are suspended. The stop-the-world monitors serve this purpose.

In more detail, a mutator regularly polls the collection flag and when it is active, it executes a synchronisation barrier. It acquires the lock of the Init monitor, increments *a global mutator counter*, wakes up the VM thread with the Init monitor, releases the Init monitor and suspends itself with the App monitor. When the VM thread wakes up, it waits until all the mutators have incremented the global mutator counter by using the Init monitor.

Then, the VM thread initialises the queue of GC tasks. After setting up the task queue, the VM thread synchronises with the GC threads. The VM thread wakes up all the GC threads that are waiting on the GC monitor, and suspends itself by waiting on the VM monitor.

2.3.2 Parallel phase

A GC thread fetches tasks from the global task queue (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 after root tasks, and are fetched by GC threads once their root tasks have finished. 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 address provided by the root task.

For each object that it 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 some root 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 avoid this, an idle GC thread "steals" a reference from a randomly-chosen BFT queue and processes the corresponding sub-graph.[3] To avoid conflicting concurrent accesses to a BFT queue, the queue owner pushes and pops from the queue head, whereas other 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 happened to pick overloaded GC threads even though they exist. To handle the second case, the GC thread does not directly leave steal mode. Instead, after a number of unsuccessful steal attempts, it enters a *termination protocol*. It first atomically increments a global thread counter, to indicate to other GC threads that it is in the termination protocol. Then, the GC thread actively polls the global 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 can end. In this case, the GC thread leaves the termination protocol. Otherwise, if the counter has not reached the number of GC threads, the GC thread "peeks" into the other BFT queues. If any is not empty, it atomically decrements the global counter and returns into steal mode; otherwise, it leaves the termination protocol.

Final task.

Once it has left the termination protocol, a GC thread 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 synchronisation aims to ensure that no GC thread is modifying the heap when the VM thread restarts.

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 them-

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

selves on the same monitor. The wake up also wakes up the other GC threads, which put themselves back to sleep again.

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. It is required only in specific configurations (out of scope here) where there is no steal task.

2.3.3 Final synchronisation phase

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

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

2.4 Old-generation collection

If an object promotion fails at any stage of young-generation collection, because no PLAB could be allocated from the old generation, this indicates that the old generation is full and needs to be collected. The GC thread that suffered promotion failure sets a flag and continues. After the parallel phase is over, if the flag is set, the VM thread starts an old-generation collection.

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

Before each phase, the VM thread initialises the task queue with appropriate tasks, then triggers the GC threads. End of a parallel phase is ensured using the same mechanism as of the young generation: with the termination protocol and with the final task. They are executed twice, one time for the marking and one time 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 implemented. This information is summarised in Table 1. Our experimental evaluation, presented in Section 4, shows that the proposed 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, it falls back to a slow path using Posix synchronisation primitives. This approach works fine for a small number of threads, as the fast path generally succeeds. However, as observed by Lozi et al. [13], on a large multicore, lock performance collapses under contention; this is particularly severe when spinning on CAS. To avoid this phenomenon, we studied 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. A lot of GC threads are waiting for a long duration, preventing them to participate to the beginning of the parallel phase, up to wait during the whole 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, we replace this with a Michael-Scott lock-free queue [16]. Recall that the VM thread adds tasks during the initialisation phase, which are fetched during the parallel phase. Thus, there is no concurrency between adding and fetching, only between fetches. Therefore, the VM thread adds without synchronisation, and GC threads fetches 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, all the GC threads reach this point at almost exactly the same time because they are synchronised before by the termination protocol. As a consequence, the lock becomes contended and its performance collapses.

Solution.

To avoid this issue, we remove the GC monitor's lock. The lock was used for two purposes: first, to protect the global thread counter used for the barrier at the end of the parallel phase; second, associated with the condition variable of the GC monitor, to suspend the GC thread.

For the former, we remove the redundant synchronisation of the final task (see Section 2.3.2). Instead of waiting for the other GC threads, the final thread simply wakes up the VM thread with the VM monitor, and then suspends itself with the GC monitor. After this change, the global thread counter is not required either.

For the latter, we replace the wait by a Linux `futex_wait` call [8]. A `futex_wait` has the semantics of an atomic compare-and-sleep, and does not require acquiring a lock. After these two changes, the GC monitor is not used anymore and we can remove the GC monitor's lock.

However, this change potentially introduces a new race condition. The final thread, while coordinating the end of the parallel phase, might be pre-empted, after waking up the VM thread but before suspending itself. It may happen that, during this time, the VM thread resumes the application,

Bottleneck	Optimisation name	Optimisation description	In NAPS
Heavily contended GC lock (3.1)	Lock-free GC task queue	GC task queue lock-free	Yes
	Lazy GC parking	GC thread sleeps lazily at end of parallel phase	Yes
NUMA heap (3.2)	Interleaved Space	Balances the load between the memory controllers	For old/permanent
	Fragmented Space	Balance the load and improve the spatial locality	For young
	Inter-node messages	Avoid remote access	No

Table 1: Optimisations implemented in NAPS

which triggers a new collection. If the ex-final thread is now scheduled by the kernel during the new parallel phase, it will suspend itself on the futex, but will never be woken up by the VM thread, leading to a deadlock.

This was not a problem in Parallel Scavenge because, during the whole execution of the final task, the final thread owned the lock of the GC monitor. Acquiring this lock is required to wake up the GC thread with the GC monitor at the end of the initialisation phase. Therefore, the final thread will inevitably suspend itself 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 at the beginning of the parallel phase. To suspend itself, the ex-final thread atomically checks with the futex that the timestamp has not been modified. If it was, the futex does not suspend the thread, which goes directly into to the new parallel phase, thus avoiding the deadlock.

3.2 NUMA heap

A NUMA machine raises two problems: load balancing between memory controllers, and latency of remote access.[4]

3.2.1 Interleaved Spaces

Issue.

When the garbage collector scans an object, this implies that it was not scanned previously in the same garbage collection cycle. Thus, a GC does not benefit from hardware data caches, and puts pressure on the hardware memory controller.

NUMA is designed to spread memory access load between multiple memory controllers. This fails in practice if the software on several cores is accessing a single controller. In particular, GC will suffer from this issue if many objects that it scans are located on the same node.

This problem concerns mainly eden space, because of the conjunction of two things: physical memory allocation policy by the Linux kernel, and sequential memory initialisation in the application, as we explain next.

Parallel Scavenge reserves a large virtual memory range for the heap using a Linux `mmap` system call. Initially, it contains no physical memory. Lazily, the first time an address in the range of a page is accessed, this generates a page fault, and the Linux kernel allocates a physical page. The allocation policy will preferentially allocate a physical page located on the node that triggered the fault [12]. Thereafter, the virtual address range remains associated to the same physical node.

Many applications start with a single-threaded initialisation phase that populates the eden space. Because of the Linux allocation policy, these objects are all allocated on the same physical memory node, and remain on that node thereafter.

This issue occurs with the other spaces and generations, but with lower probability. For instance, the application initialisation thread might create large objects, which Parallel Scavenge allocates directly in the old space. This allocation pattern would lead to an association of a large part of the virtual space to a single node.

Solution.

To improve the balance between memory controllers, we suppose that by balancing memory allocations across nodes, memory access will also be balanced. Our experiments hereafter support this hypothesis.

In an *interleaved space*, pages are located on different nodes, in round-robin: one page is located on the first node, the next page on the second one, and so on. This ensures a perfect balance, but does not help with spatial locality, because objects are allocated on random nodes.

3.2.2 Remote memory latency.

Issue.

On a NUMA machine, accessing a remote memory controller is more costly than a local one. For instance, on the Magny-Cours machine, a remote memory access takes three times the time of a local one, when the data is not in cache.

As explained earlier, the eden space is mainly allocated on one node. During single-threaded initialisation, the mutator will access local memory only, because of the Linux memory mapper. This scheme ensures a perfect spatial locality. However, in the mutator's parallel phase, its threads execute on arbitrary nodes. Therefore, they have a random probability to execute on the initial node and a random probability to access their local memory. Furthermore, access patterns in survivor, old and permanent generations are essentially random in Parallel Scavenge.

On an N-node machine, a random access has an $N-1/N$ probability of being remote, which approaches 1 when N is large. This means a high probability that accessing a live object will incur an expensive memory accesses.

Solution.

Fragmenting consists of dividing a space into multiple fragments, where a fragment is a virtual address range that is physically allocated on a single node. Technically, the memory is still lazily allocated, but the virtual address space of the fragment is associated to a given node in the kernel through a system call.

In a *fragmented space*, each TLAB or PLAB is contained within a single fragment. A thread allocates a TLAB or a PLAB from the fragment associated with the node on which the thread currently executes. Fragmented spaces

[4]Techniques similar to those discussed hereafter already existed in Parallel Scavenge, but their implementation was flawed, as discussed at the end of this section.

improve locality for both the mutators and the GC threads *(i)* because a thread mostly accesses objects that it allocated itself recently, and *(ii)* because the scheduling policy of the operating system avoids thread migration.

Fragmented space also helps for load balancing, i.e., to allocate objects equitably among the nodes, also both for mutators and GC threads because:

- The Linux scheduling policy spreads threads across the nodes whenever the number of threads of the application does not fit in a single node.
- For the mutators, we suppose that they allocate roughly the same number of bytes between two collections. This hypothesis is confirmed by our experiments.
- For GC thread, they inevitably copies almost the same number of objects thanks to the load balancing.

Therefore,a fragmented space is preferable to an interleaved one since in this case, it provides both load balancing among the nodes and locality.

3.2.3 Load-balancing vs. locality trade-off

Our analysis of load-balancing vs. locality trade-off described in the previous section is summarised in Table 2. Each line lists the properties of the corresponding space type (first column). The second column identifies whether the approach is good (+) or bad (−) for load balancing. The other columns look at 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 section (New object, Copy of object after the first collection). For GC, we distinguish the different generations, and within each generation, between reading the object (Scan) and moving it (Copy).

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/Copy). GC locality is improved by scanning the stacks of a mutator thread on the same node as that thread (Scan), and by copying objects towards the node where the GC thread runs (Copy).

We tested some other schemes aiming to improve memory locality, and observe that they degrade performance because of poor load balancing. In particular, we tested a mechanism, called "Inter-node messages," that avoids remote access by the GC. First, this optimisations restrict work stealing to the GC threads that run on the same node. Second, a GC thread running on a given node discovers a reference to an object located on the same node, it scans and copies it; but if the object is on a different node, it sends an appropriate message to a GC thread running on the other node. Somewhat counter-intuitively, our experience is that this it degrades performance significantly, compared to interleaved and fragmented spaces, because it degrades load balancing. Therefore, the evaluation section ignores this.

More generally, we find that ensuring load balance between the memory controllers is more important than improving spatial locality. Indeed, poor locality slows down the application but does not constitute a hardware scalability bottleneck. Therefore, our design avoids locality-improvement decisions that might affect memory load balance. It does not hurt to improve locality as long as memory load balance is not affected.

3.2.4 Implementation details

New heap layout.

Our design uses a fragmented space for the eden space. During the initial single-thread allocation phase, memory will not be balanced, because all objects will be allocated on a single node. But this phase is usually very short compared to total execution time. Afterwards, the fragmented eden will be beneficial if mutators allocate object at the same rate. Our evaluation validates this assumption.

For survivor spaces, we also use a fragmented space, because the GC threads copy approximately equal numbers of objects, thanks to the load balancing of the GC's parallel phase.

For the permanent and old generation, we use an interleaved space, not because the memory is not naturally equitably allocated on the nodes, but because these generations use a compacting algorithm. Zhou and Demsky [30] propose a NUMA-aware compaction algorithm, but this is out of our scope, and furthermore, our results show that the performance benefit of improving locality is small.

The resizing problem.

The new memory layout raises two issues. First, resizing a space is now substantially more expensive. As the spaces and generations are contiguous in virtual memory, resizing requires many remappings. Remapping requires two system calls, one to free the physical pages from their virtual addresses, and one to re-associate the virtual address range to different physical address. The cost increases linearly with the number of remapped pages; therefore, an interleaved or fragmented space increases resize cost proportionally to its size. Furthermore, resize occurs during the sequential phase of the collector, further increasing the impact.

Second, in a a naïve approach, 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 because a single fragment is full, even though there is still space in a different fragment; indeed, in our initial experiments, the collector was triggered much more frequently than in Parallel Scavenge. This behavior invalidates the choices made by the resizing policy.

The solution leverages the separation between virtual and physical memory. NAPS initially reserves a very large virtual space for the heap and uses counters to maintain the number of physical bytes currently allocated in each space and generation. The overhead is negligeable; on a 64-bit machine, the virtual address space is sufficiently large.

We solve the first issue by initially reserving the maximal possible heap size. This pre-allocation avoids remapping during the execution of the application.

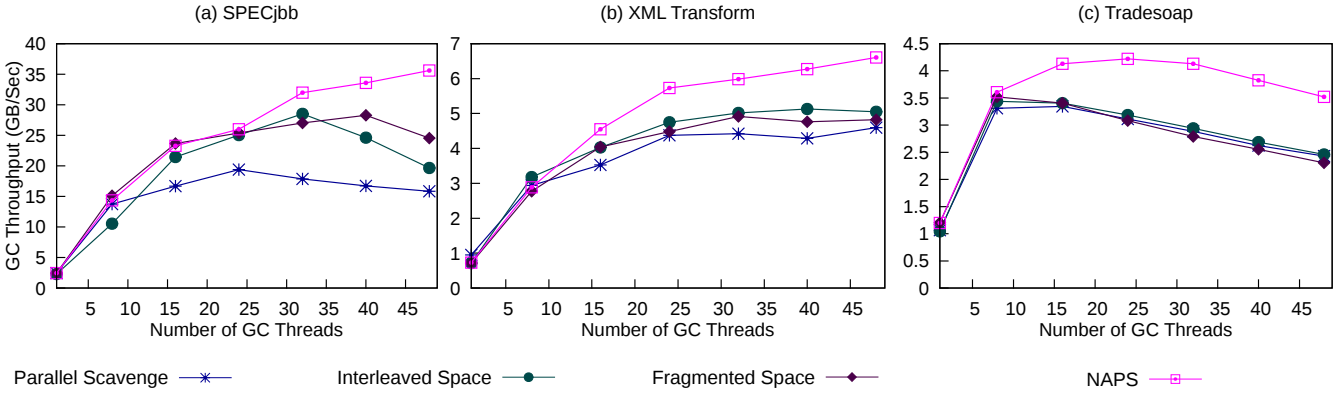
We solve the second one by making the virtual memory size of a fragment equal to the total size of the space it belongs to. This ensures that an allocation will not fail until our allocation counter reaches the assigned space size. Therefore, NAPS can trigger a collection at exactly the amount specified by the resizing policy.

Comparison with the original Parallel Scavenge code.

Parallel Scavenge already provided interleaved and fragmented spaces through a command line option. However, it did not address the cost of resizing, leading to poor performance. Moreover, the survivor space was interleaved; we find that using a fragmented space to improve GC locality gives slightly better results.

Space type	Load balancing	Mutator locality			GC locality					
		First Access	Later Access		Eden		Survivor		Old/Permanent	
			New	Copy	Scan	Copy	Scan	Copy	Scan	Copy
Parallel Scavenge	−	+	−	−	−	−	−	−	−	−
Interleaved space	+	−	−	−	−	−	−	−	−	−
Fragmented space	+	+	+	−	−	+	−	+	Not Applicable	
Inter-node messages (results not reported)	−	+	+	+	+	+	+	+	Not Applicable	

Table 2: Load balancing vs. locality properties



X axis: number of cores. Y axis: number of megabytes collected/second (higher is better)

Figure 3: Impact of each of the optimisations on the GC throughput (number of allocated bytes divided by pause time).

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

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, SPECjvm2008 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 to access physical memory. In total there are 48 cores, carrying 96 GB of RAM consisting of eight memory nodes. The system runs a Linux 3.0.0 64-bit kernel.

4.1.2 Benchmarks

We base our evaluation on applications from three benchmark suites that represent real and server-class multi-threaded workloads. In order to focus on GC, each experiment uses heap size that is very close to the application's working set size. Table 3 summarises the applications under study and their heap size.

SPECjbb2005.

SPECjbb2005 is a business logic service-side application. It emulates a three-tier client/server system, with emphasis on the middle tier [23]. This benchmark runs for a fixed amount of time (configured by the user) and measures application throughput, i.e., the number of operation 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.

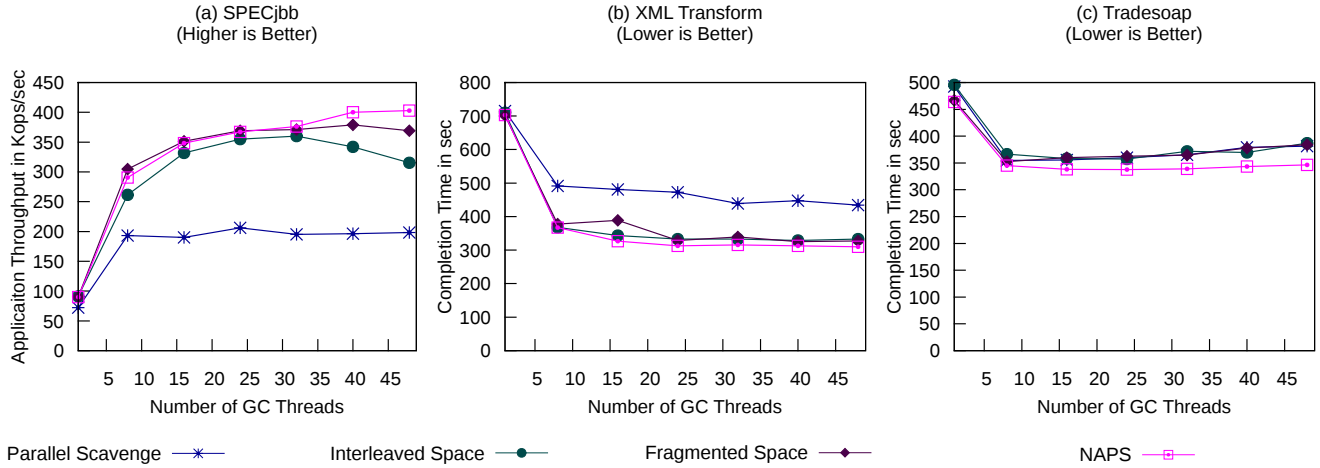
SPECjvm2008.

SPECjvm2008 is a set of real-life and area-focused applications [24]. Depending on the benchmark settings, an iteration runs either for some number of operations per mutator thread and measures the application time, or for some amount of time and measures the throughput. We discard those applications that do not generate sufficient work for the GC.

Thus this benchmark consists of: 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.

DaCapo is a set of real-life applications, aimed at eval-



X axis: number of cores. Y-axis of XML Transform and Tradesoap: completion time (lower is better); SPECjbb: throughput (higher is better).

Figure 4: Impact of each of the optimisations on application performance

uating the performance of different components of a Java Virtual Machine [4]. Each application in this suite contains a fixed, predefined-size workload. We discard most of the applications because their memory usage is insufficient to be interesting: with 48 GC threads, each GC thread has less than a few tens of kilobytes to collect.

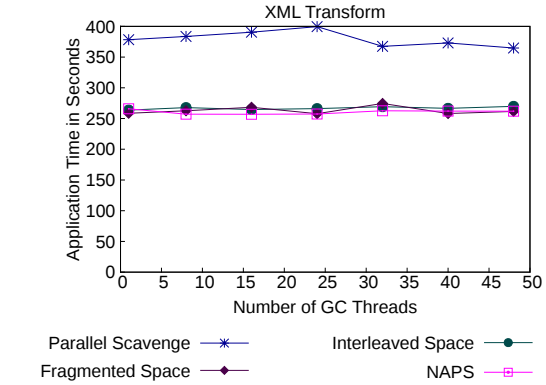
We consider only two applications in Dacapo, Eclipse and Tradesoap, as they are among the most memory intensive ones. They require a 500 Mb heap and are the less memory intensive evaluated applications.

4.2 Analysis of individual optimisations

This section discusses the performance impact of each optimisation. We focus on three applications for this analysis: SPECjbb2005, XML Transform from SPECjvm, and Tradesoap from DaCapo. We choose these three because they represent very different working sets, from small for Tradesoap to huge for SPECjbb2005. For XML Transform, we run 2 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.

We evaluate the following configurations: **Parallel Scavenge:** the baseline GC. **Interleaved Space:** Parallel Scavenge with interleaved space for all the generations. **Fragmented Space:** Parallel Scavenge with interleaved space for the old/permanent generation and fragmented space for the young generation. **NAPS:** fragmented space with lock-free task queue and Lazy GC parking. Each experiment uses 48 application threads. We measure scalability by varying the number of GC from 1 to 48, in steps of 8.

- Figure 3 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 4 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 the application in combination.
- Figure 6 plots the difference between completion time and GC pause time, in order to show the effects of our optimisations on application performance. We report curves for XML Transform only. We discard SPECjbb because we can not isolate the increase of application performance. We discard Tradesoap because it uses too little memory to show any significant result.



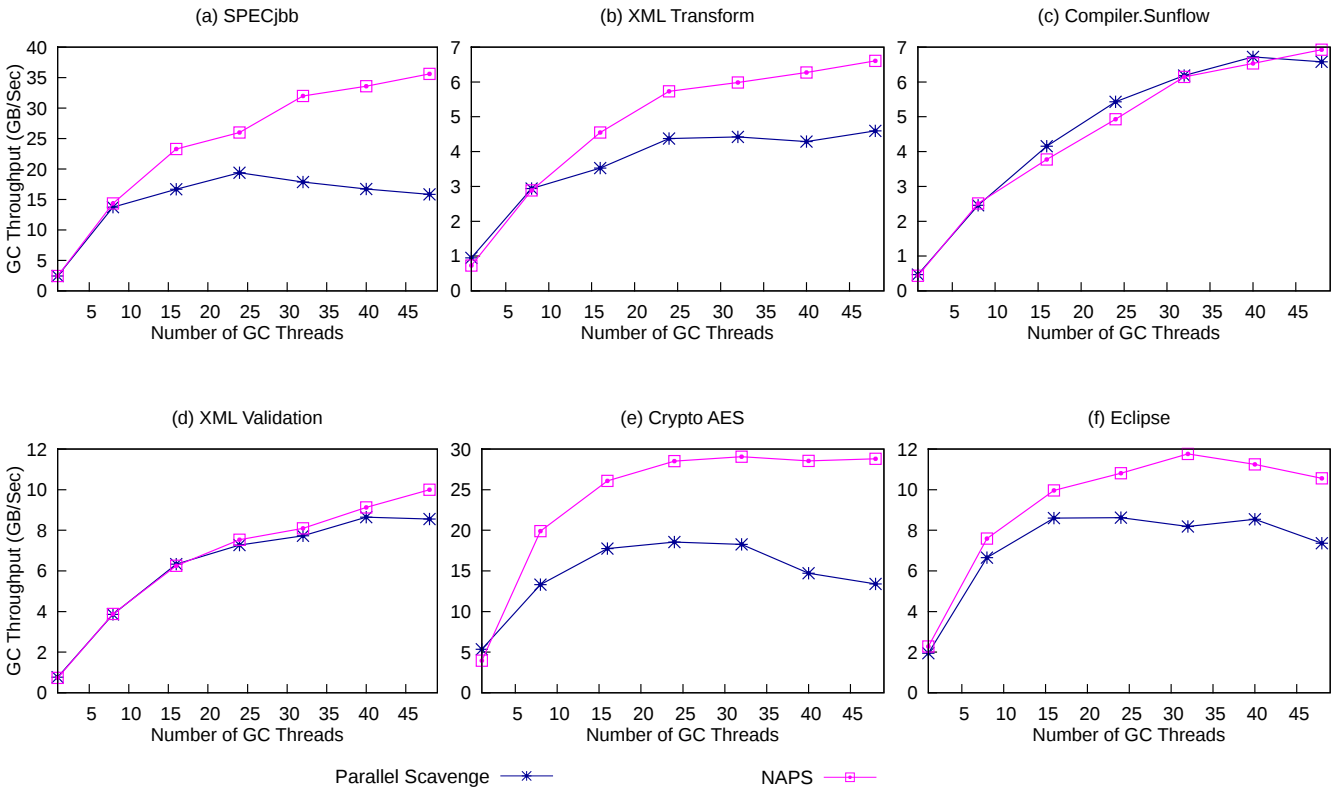
Y axis: difference between completion time and GC time (lower is better).

Figure 6: Application improvement.

4.2.1 *Interleaved space*

Interleaved spaces balance the load across memory controllers, but hurt locality.

Recall, from Section 3, that many applications initialise using a single thread, which maps the physical pages of the eden space from a single node. Indeed, with SPECjbb2005, at the time of first collection, we observe that 95% of the eden space's pages were allocated from a single node. Due to this, as shown in Figure 3.a, we observe that interleaving of young generation reduces pause time by 19% with 48 GC



X axis: number of cores. Y axis: number of megabytes collected per second (higher is better).

Figure 5: GC Throughput of Parallel Scavenge and NAPS.

threads, as compared to baseline.

XML Transform demonstrates exactly the same situation, with pause time reduced by 8% (Figure 3.b). As this application allocates less, the GC spends less time in graph traversal, and therefore, the effect of the interleaved space is less significant.

In Tradesoap, we observe that the pages are allocated mainly from two nodes in the beginning, leading to a 1% reduction (Figure 3.c). The effect of interleaved space is small, because the GC spends most of its time in synchronisation.

Impact on application. Better balance of the load across memory controllers has an impact on the application also. Figures 6 and 4.b shows this phenomenon: XML Transform runs 26% faster when using an interleaved young generation. This shows that both the application and the GC are hampered by an unbalance of the memory accesses between the nodes.

Overall impact. Overall, these results show that load balancing has a high impact on both application (Figure 4) and GC, especially if the application is memory-intensive, such as SPECjbb.

4.2.2 *Fragmented space*

A fragmented space increases the locality of the application and of the GC, while maintaining load balance between the memory controllers. As Figures 3, 4 and 6 show, improved spatial locality has only marginal impact on performance.

Impact on GC. In SPECjbb, memory locality increases GC performance (Figure 3.a) at 48 GC threads: whereas throughput degrades after 30 GC threads with interleaved spaces, it remains roughly stable with fragmented spaces to slightly degrade after 40 GC threads. We hypothesise that the difference is caused by the interconnect between the nodes that becomes a bottleneck: for any given GC thread, seven memory accesses out of eight are remote, and all the memory bandwidth of the interconnect is used. This is not related to remote latency, but to the bandwidth limitation of the interconnect.

Except with 48 GC threads in SPECjbb, a better locality increases the performance of the GC and the application only marginally. This result shows that memory latency does not impact the performance of the GC. Indeed, hardware cache line prefetechers load cache lines in advance, both in the case of sequential access and in that of a memory access decoded in the instruction pipeline. This hides memory latency successfully sufficiently often that latency is not a problem.

Impact on application. Figure 6 shows that a fragmented space has only a marginal impact on the application as well. Smaller memory latency does not change the performance of the application either.

Overall impact. Although the evaluation of fragmented spaces shows that spatial locality has only slight impact, both on application and on GC, it also shows that it should not be completely neglected, because the interconnect can also become a bottleneck. For this reason, we choose to use a fragmented space in NAPS.

4.2.3 *Lock-free task-queue and lazy thread parking*

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

Moreover, the lock-free task queue is expected to be especially significant for applications with little work for the GC, such as Tradesoap. In such cases, we have observed that, under Parallel Scavenge, some threads go straight to the end of the parallel phase of the collection, without contributing at all to the collection. The GC thread can even slow down the termination protocol. In contrast, our lock-free task-queue is expected to fetch tasks immediately, ensuring that every GC thread contributes to the parallel graph traversal phase.

Impact on GC. As expected, better synchronisations do not have much effect on memory-intensive applications such as SPECjbb2005 and XML Transform (Figure 3.a and 3.b) for a small number of GC threads. It has a greater impact on applications that do not create a lot of work for GC threads because in this case, the time to synchronise the threads become significant (Figure 3.c).

It should be noticed that for SPECjbb, the most memory intensive application, optimising the synchronisation also begin to have a significant impact after 40 GC threads. Indeed, under Parallel Scavenge, beyond 40 GC threads, the bouncing of the lock's cache line between the cores of the GC threads becomes significant and alters the performance. The NAPS lightweight synchronisation protocol avoids this overload.

Our synchronisation optimisations improve the performance of SPECjbb, XML Transform and Tradesoap by 31%, 26% and 34% respectively, over Fragmented Space optimisation when using 48 GC threads. Combined with the fragmented heap, the synchronisation optimisations make SPECjbb and XML Transform scales up to 48 GC threads. It is not the case for Tradesoap because its heap is small and adding more GC threads only increase the pressure on the cache lines that contain the objects.

Impact on Application. Figure 6 shows that, beyond GC pause time improvement, improved GC synchronisation does not noticeably impact application performance.

Overall impact. As compared to baseline, the pause time reduction due to these two optimisations translates to a total application time improvement of Tradesoap and XML Transform of 9% and 28% respectively, and improved throughput of SPECjbb2005 by more than two times, at 48 GC threads (see Figure 4).

4.3 Scalability

Figure 5 compares throughput of Parallel Scavenge and NAPS, i.e., total amount of memory allocated during the whole execution, divided by total pause time.[5] It is unfair to compare throughput across applications, since GC performance depends on the number of allocated bytes, but also on the number of objects that survives their collection.

Observe, that NAPS improves performance and scalability over Parallel Scavenge in all cases. This shows that the identified bottlenecks are real.

[5]The results for SPECjbb2005 and XML Transform for NAPS and Parallel Scavenge are common with the Figure 3 that presents the in-depth analysis of each of the optimisations.

	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 4: Individual pause time

Observe also The second observation is that by adding more GC threads, the performance of NAPS never degrades. For Crypto AES NAPS stops scaling after 24 and for Eclipse it degrades after 32 GC threads. For these applications, the number of objects that survive their collection is small and therefore, adding more GC threads does not decrease the collection time.

For the other applications, in particular SPECjbb, which is the most representative server-class application, NAPS continues to scale up to 48 GC threads.

Finally, Table 4 reports, for each of the applications, the pause time of a single collection pause. NAPS always reduces this individual pause, by up to 2.8 times in the best case. This results shows that using a scalable stop-the-world collector increase the responsiveness of the application. Moreover, a collection never pauses the application for more than 100ms.

To summarise, these results show that a stop-the-world garbage collector, such as NAPS, is able to scale roughly linearly up to 48 GC threads with an admissible pause time, as soon as it has enough memory to collect.

5. STATE OF THE ART

There has been much research on concurrent garbage collection, mainly targeting real-time applications [5, 15, 20, 21]. These collectors have been evaluated on eight cores and two nodes at most. The evaluation of the recent stop-the-world collector Immix has the same limitation [3]. This is insufficient to expose the problems that arise in large multicores. For intance, Gidra et al. [9] show that Garbage First [5] collapses after 20 cores.

Zhou and Demsky [30] propose a new parallel mark-and-compact collector targeting the TILE-Gx microprocessor family, for which each core is a memory node [28]. 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 the memory controller is the bottleneck, not 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 [18] and Tikir and Hollingsworth [27] ensure that an object is copied to the memory node where it is accessed most of the time. These algorithms increase 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. [7] propose a per-object work stealing technique using lock-free double ended queues, also used by Marlow et al. [15]. 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, hence balances the load across the memory controllers. Oancea et al. [17] 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 increase locality and concurrency is to use thread-local heaps [1, 6, 14, 22, 25]. 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 on the 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. [22] is applied, applications must still pay the price of a write barrier.

Iyengar et al. [10] 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 grows. Our results do not support this statement; on the contrary, well-established parallel computing techniques appear 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 solve these bottlenecks using well-established parallel programming techniques, and show that it was not caused by the stop-the-world design but by some of the mechanisms that were not planed for contemporary NUMA multicore with tens of cores. Our evaluation suggests that, currently, there is not any conceptual reason to think that the pause time of a stop-the-world GC should increase with the increasing number of cores and memory of multicore hardware. It leads to the conclusion that the costly and complex fine-grain synchronisations of concurrent GCs is not yet required, both to achieve good performance and to scales with the increasing number of cores and size of the heap.

As a future work, we propose to adapt the number of GC threads to the actual workload of the GC. Indeed, as reported in Section 4.2 using too many GC threads on a small heap leads to useless GC threads that increase the contention on the objects and degrade the performance.

Availability

NAPS is publicly available under an open-source license at the URL: URL-hidden-for-blind-reviewing.

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

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

[7] 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.

[8] 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.

[9] 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.

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

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

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

[19] OpenJDK Memory. Memory management in the Java hotspot^{TM} virtual machine. Technical report, Sun Microsystems, 2006.

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

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