A Performance Study of Java Garbage Collectors on Multicore Architectures

Maria Carpen-Amarie
Université de Neuchâtel
Neuchâtel, Switzerland
maria.carpen-amarie@unine.ch

Patrick Marlier
Université de Neuchâtel
Neuchâtel, Switzerland
patrick.marlier@unine.ch

Pascal Felber
Université de Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Gaël Thomas
Telecom SudParis
Évry, France
gael.thomas@telecom-sudparis.eu

ABSTRACT

In the last few years, managed runtime environments such as the Java Virtual Machine (JVM) are increasingly used on large-scale multicore servers. The garbage collector (GC) represents a critical component of the JVM and has a significant influence on the overall performance and efficiency of the running application. We perform a study on all available Java GCs, both in an academic environment (set of benchmarks), as well as in a simulated real-life situation (client-server application). We mainly focus on the three most widely used collectors: ParallelOld, ConcurrentMarkSweep and G1. We find that they exhibit different behaviours in the two tested environments. In particular, the default Java GC, ParallelOld, proves to be stable and adequate in the first situation, while in the real-life scenario its use results in unacceptable pauses for the application threads. We believe that this is partly due to the memory requirements of the multicore server. G1 GC performs notably bad on the benchmarks when forced to have a full collection between the iterations of the application. Moreover, even though G1 and ConcurrentMarkSweep GCs introduce significantly lower pauses than ParallelOld in the client-server environment, they can still seriously impact the response time on the client. Pauses of around 3 seconds can make a real-time system unusable and may disrupt the communication between nodes in the case of large-scale distributed systems.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.4.2 [**Software**]: Garbage collection

Keywords

Performance analysis, multicore, garbage collection

PMAM '15, February 7-8, 2015, San Francisco Bay Area, USA

http://dx.doi.org/10.1145/2712386.2712404.

1. INTRODUCTION

Nowadays, multicore systems are the norm for standalone servers or components of big server farms. The performance of modern computers will continue to improve with the increasing number of cores per CPU. Today, server applications often execute on language runtime, for example with JBoss, Spark or Neo4j. A critical part of any language runtime is the Garbage Collector (GC). A significant effort has been put into optimizing garbage collectors for multicores [15, 18, 11, 16], in order to reduce the application stalls. However, recent studies show that current GCs do not scale well with the number of cores [12] and that they can generally degrade the application responsiveness or throughput.

In this work we study to what extent can the GC affect the application performance and give some pointers on what configurations can have an important impact on the application execution. We assess the performance in terms of both responsiveness (i.e., the delay between the generation of an event and its acknowledgement) and throughput (i.e., the amount of work performed in a given period of time). We also measure the duration of the GC pause times, i.e., the period of time when none of the application threads is progressing. We experiment with the OpenJDK8 Java Virtual Machine (JVM), being one of the most extensively used language runtimes. All collectors in JVM are generational; this means that the heap is split in two parts: the Young and the Old Generation. All allocations are made in the Young Generation, and the objects that survive a few collections are moved to the Old Generation. As an optimization for multicore systems, a Thread Local Allocation Buffer (TLAB) is used in the Young Generation (i.e., chunks of the Young Generation are pre-allocated for each thread, which subsequently takes care of the local allocations). Thus, most of the time there is no need for synchronization support when allocating new objects. We exhaustively test the efficiency of all current GCs both in an academic environment (with a set of benchmarks in the DaCapo suite), as well as in a real-life scenario (running the massively scalable database Apache Cassandra). The benchmark experiments are split in two categories: (1) when a system (full) GC is forced between the benchmark iterations and (2) when no system GC takes place and the memory is collected only when needed.

Overall, our study tends to show that sometimes the GC might have an unexpected behaviour (e.g., longer pauses for smaller Young Generation size, TLAB decreasing per-

formance in some cases, etc.). In detail, we found that:

- **Total execution time:** we measure the total execution time for the benchmarks in our subset with different GC and heap sizes. G1 GC has the worst throughput amongst the GCs for test case (1), while for (2), the GCs perform in a similar manner.
- **GC pause time:** we study the length of the application pauses caused by the GC activity, when the heap and the Young Generation sizes vary. We observe that G1 GC performs the worst in situation (1) for all tests.
- **GC statistics**: even though the Serial GC has no synchronization mechanisms, we find that it gives the best execution in less than one fourth of the tests when there are no GC pauses at all. Furthermore, for ConcurrentMarkSweep and ParNew GCs, we observe that sometimes a smaller Young Generation size results in a bigger *average pause time* for the same heap size.
- **TLAB influence:** we checked if enabling the TLAB leads to a performance improvement for each GC and benchmark. We considered a variation of 5% from the average total execution time. We find that most of the time the TLAB does not have any influence (either good or bad), but sometimes it even degrades the performance (e.g., for G1 and *pmd* benchmark).
- **GC ranking:** we order the GCs according to the best execution times of the benchmarks in our experiments. Based on this simple ranking and the previous tests we find that the default GC, ParallelOld, is constantly performing well, while both G1 and ConcurrentMarkSweep tend to degrade the application performance.

For the client-server experiments we used Cassandra DB as the server and the YCSB benchmark on the client-side. We discuss the impact of three most efficient GCs on multicore machines (ParallelOld, ConcurrentMarkSweep and G1):

- **on the server side:** a detailed analysis of Cassandra's logs indicates application pauses of up to 4 minutes for ParallelOld, and of 3-5 seconds for G1 and ConcurrentMarkSweep. Even though the latter is much smaller than the former, it is not negligible in systems where the responsiveness is critical.
- **on the client side:** we study the client response time and observe that most of the peaks in the response time correspond to the moments when a GC took place. The result shows that the server is unable to progress during the pause time. This degrades the user experience and, possibly, the progress on other nodes (that are either waiting for the response or suspect that the collecting server is faulty).

Based on these observations, we draw the conclusion that there are workloads that are executed better with particular GCs and sometimes the assumptions on the GC activity do not hold true for all scenarios. We show that on the benchmark suite Java's default GC gives the best overall performance, while on a memory-intensive database server, it introduces unacceptable pauses and affects the response time on the client. Moreover, even the concurrent low-pause GC, ConcurrentMarkSweep, and G1 GC that ensures bounded pause time, end up stopping the application threads for a few seconds. This is still a significant pause that can affect the execution of the application.

2. BACKGROUND

The **Garbage Collector** (GC) represents a critical element in any managed environment, such as Java Virtual Machine (JVM). Its purpose is to automatically reclaim unused memory and protect the application from accessing invalid references. In order to do that, it must regularly stop all the application threads in order to avoid concurrent accesses to the heap or the execution stacks between the GCs and the application. This happens in single core, as well as in multicore environments.

Typically, the garbage collectors in Java are *generational* [17]. The generational algorithms are based on two observations: most allocated objects are not referenced for long (they die young) and there are few references from older to younger objects. All objects are allocated in the Young Generation area; if they survive a number of collections, they are copied to the Old Generation. When the application receives an "Allocation Failure" message, it performs a garbage collection of the Young Generation. The Young Generation collections are fast and efficient, since the collected space is small and likely to have a lot of objects that are no longer referenced. The surviving objects are promoted to the Old Generation. If there is no more space in the Old Generation to bring new objects, then the GC executes a *major* (or *full*) garbage collection, trying to free space in all generations.

In general, an application running in a managed environment will encounter multiple safepoints during its execution. A **safepoint** is a step in the program execution when all threads running Java code need to be stopped for a period of time. This is also called a **stop-the-world** pause. Stopping all threads is necessary for the initiator of the safepoint to get exclusive access to the JVM data structures. There are multiple reasons that could trigger a safepoint: garbage collector pauses, code deoptimization, flushing code cache, biased lock revocation, etc.

Table 1 represents the list of the GCs available in Java, with a description of their components for the Young and Old Generations. The only GC that does not use a parallel collection for the Young Generation is SerialGC. In its case, both generations are collected sequentially, on a single thread. However, SerialGC is the simplest GC, requiring no synchronization mechanisms for allocating memory. The ParNew GC and Parallel GC both use a single-threaded collection for the Old Generation. For the Young Generation collection they take advantage of the same parallel algorithms as the ConcurrentMarkSweep GC, and respectively the ParallelOld GC. The main difference between these two parallel algorithms is that one of them was especially created to work with ConcurrentMarkSweep (besides the Young Generation collection, it also takes care of the synchronization needed during the concurrent phases of ConcurrentMarkSweep in the Old Generation). Finally, the newest GC, called Garbage-First, is a parallel collector for both generations that can ensure a bounded pause time.

2.1 DaCapo Benchmarks

The DaCapo benchmark suite [3] is a memory-management benchmarking tool for Java. It consists of the following set of open source, real world applications:

Table 1: List of garbage collectors used in this work and their characteristics

GCs	Young				Old			
	Parallel	Copying	Concurrent		Parallel	Compacting	Concurrent	
			Marking	Copying			Marking	Compacting
Serial	No	Yes	No	No	No	Yes	No	No
ParNew	Yes	Yes	No	No	No	Yes	No	No
Parallel	Yes	Yes	No	No	No	Yes	No	No
ParallelOld	Yes	Yes	No	No	Yes	Yes	No	No
CMS	Yes	Yes	No	No	Yes	No	Yes	irrelevant
G1	Yes	Yes	No	No	Yes	Yes	Yes	No

- **avrora:** single external thread, but internally multi-threaded.
- **batik:** Mostly single-threaded both externally and internally.
- **eclipse:** single external thread, internally multi-threaded.
- **fop:** single-threaded.
- **h2:** multi-threaded (one client thread per hardware thread).
- **jython:** single external thread, internally using one thread per hardware thread.
- **luindex:** single external thread, internally it uses some helper threads to a limited degree displaying limited concurrency.
- **lusearch:** multi-threaded, one client thread per hardware thread.
- **pmd:** single client thread, internally multi-threaded using one worker thread per hardware thread.
- **sunflow:** multi-threaded, driven by a client thread per hardware thread.
- **tomcat:** multi-threaded, driven by a client thread per hardware thread.
- **tradebeans:** multi-threaded, driven by a client thread per hardware thread.
- **tradesoap:** same as tradebeans.
- **xalan:** multi-threaded, driven by a client thread per hardware thread.

When executing a DaCapo benchmark with the default options, it only runs once, returning the execution time at the end. However, a number of parameters can be fed to DaCapo. An important option is the number of iterations that the benchmark will execute. By default, the application will perform a system GC between every two iterations. This feature can also be disabled. When running multiple iterations, all except the last one represent warm-up rounds; the last iteration is the actual run of the benchmark. Besides this property, one can also indicate the number of threads for the current execution. This option will overwrite the default setting of having one client thread per hardware thread.

2.2 CassandraDB

Cassandra [2] is a distributed on-disk NoSQL database, with an architecture based on Google's BigTable [6] and Amazon's Dynamo [8] databases. It provides no single point of failure, and is meant to be scalable and highly available. Data is partitioned and replicated over the nodes. Durability in Cassandra is ensured by the use of a commit log where all the modifications are recorded. Exploring the whole commit log to answer a request is expensive; thus, Cassandra also has a cache of the state of the database. This cache is partially stored to disk and partially stored in memory. After a crash, a node has to rebuild this cache before answering client requests. For this purpose, it rebuilds the cache that was stored in memory by replaying the modifications from the commit log. Through a couple of configuration files, one can select the GC, heap and Young Generation sizes and modify the amount of memory used for the cache and for the commit logs.

A good way to test the responsiveness of the database is to run a benchmark on the client-side, such as Yahoo Cloud Serving Benchmark (YCSB) [7]. YCSB Client is a workload generator, having predefined core workloads that can be further extended according to the user's needs. The workloads define the number of *read*, *insert* and *update* operations executed on the database, the number of records that will be affected, the execution time, etc. It can be used in two states: the loading phase and the transactions phase. The former only populates the database (loads the data), while the latter executes the specified workload against the database. At the end, it returns statistics about the execution.

3. BENCHMARK APPLICATIONS

3.1 Experimental Setup

We perform the experiments on a 48-core server with 64GB RAM, running Ubuntu Linux on 64 bits. The cores are distributed over 4 sockets: 2 NUMA nodes per socket, each having 6 CPUs. Each core benefits of a 1.5MB Level-1 cache (different for instructions and data) and a 6MB Level-2 cache. A 12MB Level-3 cache is available per NUMA node. We use OpenJDK8 for all the tests. We considered as **baseline** the default GC configuration used by Java: ParallelOld GC, with a maximum heap size of ∼16GB, maximum Young Generation size ∼5.6GB and TLAB enabled. We set both the minimum and maximum heap size at the same value, so that we have a fixed heap size.

Starting from this configuration, we ran benchmarks of

the DaCapo suite with all the supported GCs. We varied the maximum heap size from the baseline to the maximum amount of memory supported by the machine, i.e., 64GB. Separately, we varied the Young Generation size from the baseline to the heap size. Finally, we tested separately when the TLAB is enabled or disabled. Typically, we used the default DaCapo configuration for the benchmarks, that takes advantage of the maximum number of hardware threads available on the machine. We generally configured 10 iterations for each benchmark. We also switched between having a system GC after each iteration and no system GC inserted by the benchmark.

3.2 Subset of Benchmarks

There are 14 benchmarks in the most recent DaCapo suite, released in 2009. Out of these, 3 benchmarks crashed on every test: *eclipse*, *tradebeans* and *tradesoap*. For other benchmarks, e.g., *avrora*, the execution time from one iteration to the next varied significantly. Thus, in order to identify a subset of stable benchmarks, we ran each 10 times, with the baseline Java configuration. A run consisted of 10 iterations. After each iteration a system GC takes place. We take as metrics for stability the following two characteristics:

- **The duration of last iteration:** we do not take into consideration the duration of warm-up rounds, expecting the actual run duration to be stable.
- **The total execution time:** even though the duration for each iteration varies separately, we check if the total execution time, i.e., the sum of the durations of all iterations, remains constant.

Based on the information gathered we selected the benchmarks listed in Table 2. All other benchmarks in the DaCapo group showed variations of more than 5% in both measured times: execution time and final round. We accepted in our experiments the benchmarks that are stable for at least one characteristic. In the rest of this section we will only focus on the selected subset of benchmarks.

Table 2: Relative standard deviation for the total execution time and final iteration for a subset of DaCapo benchmarks

Benchmark	Final iteration (%)	Total execution time (%)
h2	1.8	1.2
tomcat	1.8	1.2
xalan	6.4	4.2
jython	5	3
pmd	1.1	0.8
luindex	2.8	4
batik	11.2	3.6

3.3 GC Pause Time

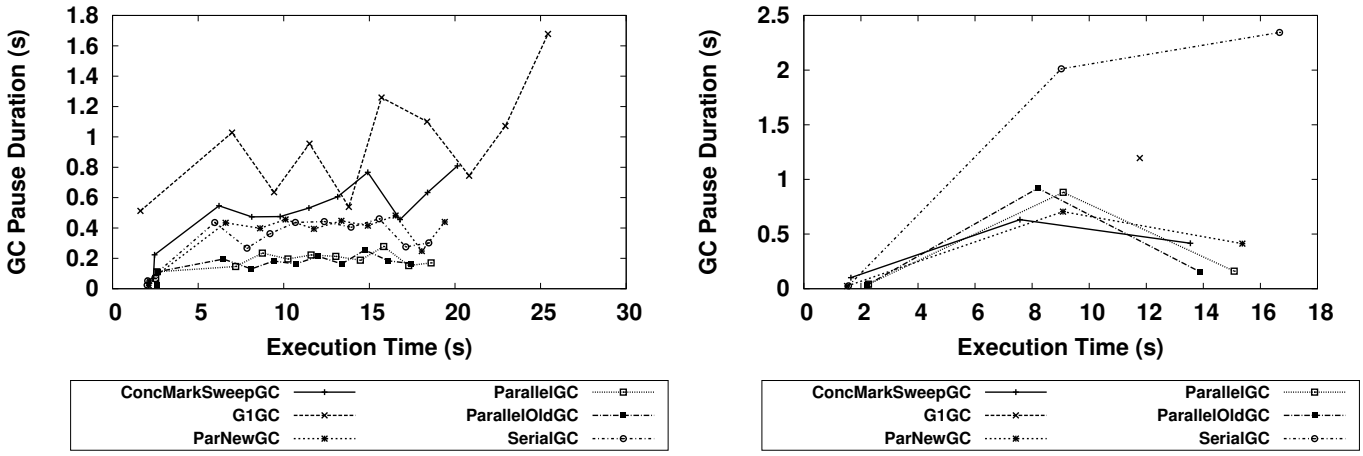
All currently implemented GCs in OpenJDK8 need to stop the application threads in at least one of their collection phases. We compare the pauses caused by GCs for the selected benchmarks. Figure 1 shows the application pause durations for all supported GCs when executing the Xalan benchmark. We choose Xalan for clarity, all other benchmarks having a similar behaviour. On both charts of the figure, the X axis represents the total execution time of the benchmark in seconds, while the Y axis indicates the duration of the pause in seconds. Figure 1(a) illustrates the application pause times when the system GC, which forces a full collection, is activated between iterations, while in Figure 1(b) this feature is deactivated (i.e., there are full GCs only when needed). Both tests were conducted with the baseline configuration for the heap/Young Generation size and TLAB. We observe that the G1GC performs the worst when it is forced to have multiple full collections, both in terms of pause duration and execution time, which can be 25% longer than for all the other GCs. In the example showed by this benchmark, one of the best performing GCs is ParallelOldGC, which is also the default GC for Java. In contrast, Figure 1(b) shows only one pause generated by G1 (the singular point at 12 seconds of execution) and the worst performance is given by the SerialGC in this case.

In order to further assess the efficiency of the existing GCs we also checked their performance relatively to the execution time of the selected subset of benchmarks. We illustrate the results in Figure 2. The charts show only the last 6 iterations, considering that the first 4 warm-up rounds are enough for the benchmark execution to stabilize. As for Xalan, the G1GC is clearly the slowest when forced to do full collections at every iteration. In the final iteration, which corresponds to the actual benchmark run, ParallelOld has the best execution time, G1 the worst and ParallelGC the second worst, since its full collections are serial. However, when the system GC is not imposed, G1 has one obvious spike, which corresponds to the only performed collection. In the last iteration, all GCs perform similarly in this case.

Next, we evaluated the correlation between the pauses caused by the GCs and the sizes of the heap and the Young Generation. We expected to find the relation described by Blackburn et al [4, 5]: the **total pause** increases with the decreasing Young Generation size, in applications that spend most of the time in Young Generation collections. This happens because the total number of minor collections increases, when the Young Generation is smaller. However, the **average pause** should decrease with the decreasing Young Generation size.

First, we analyzed the relation of the GCs when there are no pauses at all. As an example, the Batik benchmark did not perform any garbage collection with our aforementioned heap configuration and with the system GC disabled. In this particular case, the SerialGC should have the lowest execution time, since it is the simplest implementation that does not need synchronization. However, for a 64GB heap with a 12GB Young Generation size, it performs worse than all other GCs. Moreover, in only 4 out of 18 experiments, the total execution time of the SerialGC is the best.

Given the fact that some benchmarks did not show any GC with our heap configuration, we ran the experiments once more with the following heap sizes: 1GB, 500MB, 250 MB, and Young Generation sizes: 200MB, 100MB. Based on both sets of experiments, we observed that our expectation does not hold in all cases. We can take as an example the H2 benchmark and its results for the ConcurrentMarkSweep collector (Table 3). In the upper side of the table we keep the heap size constant, at 64GB and vary the Young Generation size from 6GB to 48GB. We observe that the **average**



(a) System GC

(b) No System GC

Figure 1: GC pause time for the Xalan benchmark with and without a system GC between iterations

pause duration for the smallest Young Generation size is significantly longer than for a bigger Young Generation size. Likewise, the average pause duration for a Young Generation of 24GB is longer than the one for 48GB. In the second part of the table, we list the results for the small heap and Young Generation sizes. Having such a small amount of memory for this benchmark results in hundreds of garbage collections during the execution. It is interesting to point out that in the case of a small heap the total pause time can represent more than 50% of the total execution time. Another important fact is that the ParallelOld collector behaved as expected in both situations.

Table 3: Statistics for the H2 benchmark with different heap and Young Generation sizes

Heap-YoungGen size	#pauses (full)	AVG pause time(s)	Total pause time(s)	Total execution time(s)
64GB-6GB	4(0)	1.33	5.34	196.23
64GB-12GB	2(0)	0.46	0.92	193.45
64GB-24GB	2(0)	0.55	1.11	193.31
64GB-48GB	2(0)	0.36	0.72	193.51
1GB-200MB	68(1)	0.07	4.53	192.39
1GB-100MB	136(1)	0.05	7.18	192.98
500MB-200MB	74(7)	0.13	9.78	193.19
500MB-100MB	135(3)	0.05	6.86	193.53
250MB-200MB	655(356)	1.05	689.72	1112.51
250MB-100MB	380(324)	1.33	503.89	788.43

3.4 TLAB Influence

The **Thread Local Allocation Buffers (TLAB)** represent chunks of Young Generation, one buffer per thread, where the new objects are first allocated. This allows for faster memory allocation, since the thread is able to allocate memory inside the buffer without a lock.

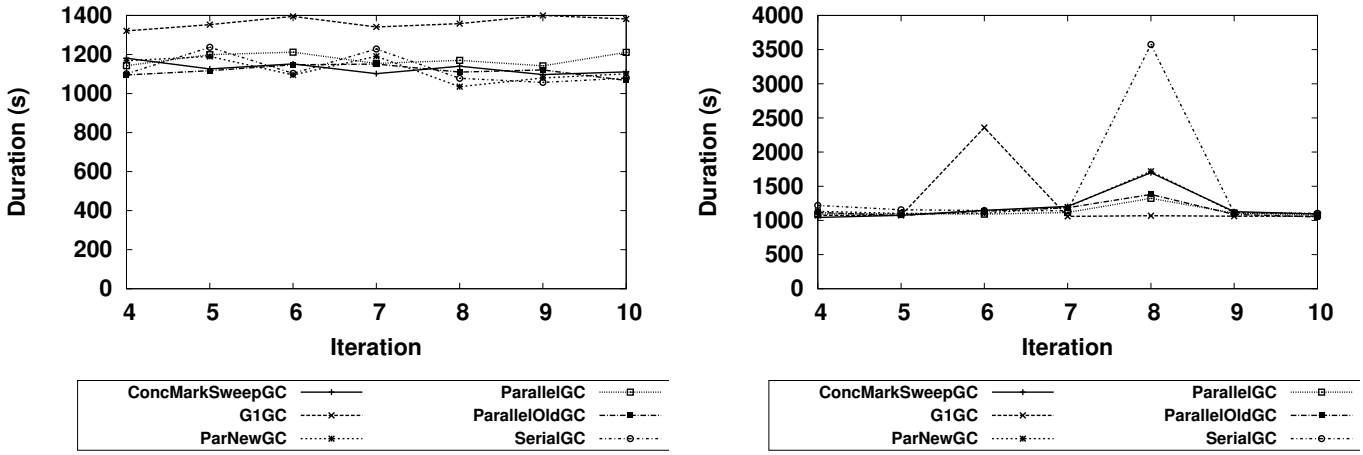
We considered that the TLAB has a positive impact for a GC when the total execution time for that GC was smaller than the same experiment without TLAB. To account for the variation in the execution times, we computed a 5% deviation from the average execution time. Thus, if the difference between the total times with and without TLAB is included in the interval [-deviation,deviation], we say that enabling the TLAB does not bring either improvement or deterioration (=). If the total execution time without TLAB is greater than the execution time with TLAB (plus the deviation), it means that enabling the TLAB results in an improvement in the execution time (+). Otherwise, we mark it as negative influence (-).

Based on the results obtained for the baseline configuration of the heap and Young Generation size, we observe that in most cases the TLAB does not have a particular influence (Table 4). However, there are cases when enabling the TLAB leads to a decreased performance (e.g., for G1 and ParNew). We observe that for the Xalan benchmark, the TLAB is at most indifferent, creating a deterioration for G1, ParallelGC and SerialGC.

3.5 GC Ranking

In order to have an overall idea on the GC influence over the DaCapo benchmarks, we finally classified them according to the number of experiments in which they performed the best. An experiment is defined by a benchmark, a heap size and a Young Generation size. For each experiment we consider the run with the shortest execution time as the best. The Y axis in Figures 3(a) and 3(b) shows the percent of experiments for which each GC was employed in the best run. When the system GC is enabled between all iterations (Figure 3(a)) there is no column for G1 GC. That means that G1 did not perform better than all other GCs in any of the experiments. If we disable the system GC (Figure 3(b)), G1 GC improves, but it is still the worst according to this metric. On the other hand, we observe that the default GC, ParallelOld, has a good performance in both cases, contributing to more than 20% of the best execution times with system GC enabled and almost 30% otherwise.

Based on these results, we conclude that the ParallelOld



(a) System GC

(b) No System GC

Figure 2: Execution time for the Xalan benchmark per iteration

Table 4: TLAB influence over all GCs and the selected subset of benchmarks

Benchmark	ConcMarkSweepGC	G1GC	ParNewGC	ParallelGC	ParallelOldGC	SerialGC
batik	+	=	+	=	-	=
h2	=	=	=	=	=	=
jython	=	-	-	+	=	=
luindex	=	+	-	=	=	-
pmd	=	=	=	=	=	=
tomcat	=	=	=	=	=	=
xalan	=	-	=	-	=	-

GC is a fitting choice for a default GC, since it proves to be stable and has good results. Running either of the concurrent collectors, G1 and ConcurrentMarkSweep, resulted in both longer pause times and execution time. The next section debates the impact of these three most important GCs on a memory-intensive real-life application. We show that in this case, ParallelOld has significant drawbacks, as compared to the other two collectors.

4. CLIENT-SERVER APPLICATIONS

First, we analysed the impact of garbage collector activities on a set of benchmarks. The next step is to check if the previous results hold on a real-life environment, such as a client-server system. Our experiments follow two main directions:

- the duration of the GC-induced pauses on the server
- the impact on the response time at the client side

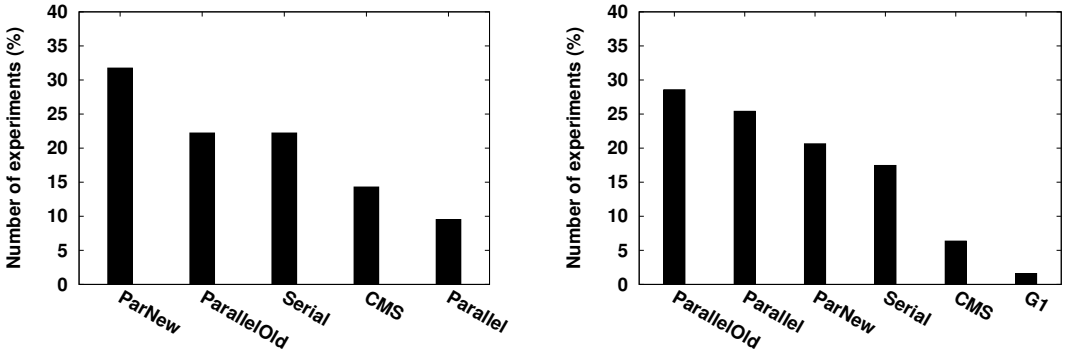
For both experiments we use Cassandra DB on a single node as the server in combination with the YCSB benchmark [7] as the client. We ran Cassandra 2.0.0 on a 48-core machine, with 64GB RAM memory, and YCSB on a 16-core machine, 8GB RAM. On the server we set the heap size at 64GB and the Young Generation size at 12GB. This experiment also aims to evaluate the behaviour of the GCs with a very big heap and Young Generations size. In the previous section we were unable to perform this test, since for these values the DaCapo benchmarks were not filling up the memory and there were no GCs. We chose this size for the Young Generation according to the JVM recommendations to be around one fourth of the total memory.

4.1 GC Impact on Server Side

We first study the performance of the default GC, ParallelOld. We based our main experiments on two different Cassandra configurations. In both cases, the YCSB client is used in the *loading phase*, i.e., it continuously populates the database with records, for a specified amount of time.

- **Default configuration**: We ran the experiment with 100 client threads in parallel for one hour and two hours, respectively. The shorter test case ends up with no full GC; nonetheless the collection of the Young Generation reaches a peak pause of around 17 seconds. The latter stressed the memory even more by loading records for an extra hour. This resulted in a full GC that stopped the application threads for more than 160 seconds. Moreover, the Young Generation collections take up to 25 seconds.

- **Stress test configuration**: We took advantage of Cassandra's internal data structures and configured it to flush as rarely as possible its records to disk and stress the memory until the server is saturated. That is, we set up both the commitlog and the internal caching structure of Cassandra (called *memtable*) to



(a) System GC

(b) No System GC

Figure 3: GC ranking according to the number of experiments in which they performed the best

have the same size as the heap, which means that everything was always kept in memory. Moreover, we already loaded the database with records so that when it starts the memory would be already partially occupied. Then, we ran the same workload as before for two hours. This experiment results in a full GC lasting around 4 minutes.

Finally, we experiment with ConcurrentMarkSweep and G1 on the same heavy workload with the stress test configuration of Cassandra. Figure 4 shows the resulting pauses caused by the GC activity. The X axis indicates the execution time. Even though the YCSB client itself ran for a fixed amount of time (two hours), the total execution time in the chart is longer. It also contains the loading step of Cassandra: because of our configuration, the server must first bring into memory all commitlogs and replay already executed transactions. Only then we start the actual benchmark. We observe that the executions of ConcurrentMarkSweep and G1 are comparable in terms of stop-the-world pause duration. Both of them reach pauses of more than 2 seconds, going up to 3.5 seconds for G1.

Even if the pauses caused by ConcurrentMarkSweep and G1 are considerably smaller than those resulted from the use of ParallelOld, they can still be unacceptable in certain circumstances. Apache Cassandra is a distributed database, supposed to run on multiple nodes for an indefinite amount of time. In only two hours we succeeded to saturate the server and obtain garbage collections lasting for a few seconds with G1 and ConcurrentMarkSweep, respectively for a few minutes with ParallelOld. Moreover, in a distributed system, even a lag of a few seconds might result in the current node being considered down and the initiation of a cumbersome synchronization protocol.

4.2 GC Impact on Client Side

We used a custom workload for the client-side experiments: 50% *read* and 50% *update* operations. We ran the experiment three times, with the three main garbage collectors supported by Cassandra: Parallel Old, Concurrent MarkSweep and G1.

The charts resulted from the above experiment are illustrated in Figure 5. In order to make the charts more readable and to reduce the size of the images, we only plotted

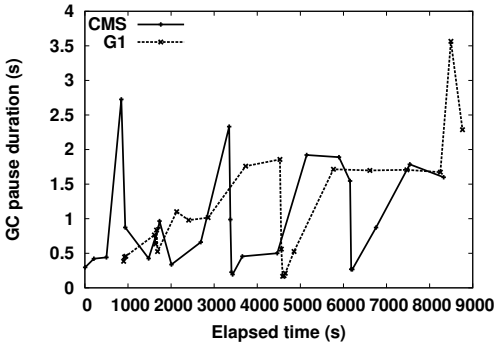


Figure 4: Application pauses for ConcurrentMarkSweep (CMS) and G1 garbage collectors with Cassandra DB server

the highest 10000 points of every chart. The X axis represents the execution time passed since the beginning of the experiment, in seconds. The Y axis indicates both the latency of the aforementioned operations and the GC pause length, during the experiment. We make two important observations:

1. most of the points are following a well defined low latency line; moreover, for the *update* operations the line of points is constant for all three GCs, while for the *read* operation, the line has some increasing "steps". Apart from the constant line of values, there are the spikes that we expected to see: a few tens of points with greater latency, for both kinds of operations.

2. by plotting the GC pause duration on the same chart as the operation latencies, we observe that the highest latencies correspond to the moments when a collection took place. Thus, we conclude that the latency peaks are strongly related to the garbage collection activity.

In order to further analyse the aforementioned results, we computed the following for each of the 3 experiments with different GC strategies:

- the average latency

Table 5: Statistics regarding the average latency for READ and UPDATE operations for ParallelOld GC

	READ	UPDATE
AVG(ms)	4.875	0.993
MAX(ms)	372.361	229.155
MIN(ms)	0.644	0.545
0.5x-1.5x AVG (%reqs)	40.412	98.639
0.5x-1.5x AVG (%GCs)	0.0	0.0
>2x AVG (%reqs)	6.895	1.065
>2x AVG (%GCs)	100.0	100.0
>4x AVG (%reqs)	6.412	0.604
>4x AVG (%GCs)	100.0	100.0
>8x AVG (%reqs)	6.374	0.09
>8x AVG (%GCs)	100.0	100.0
>16x AVG (%reqs)	0.002	0.008
>16x AVG (%GCs)	94.595	100.0

- the percent of points and the percent of GCs between 0.5x and 1.5x the average
- the percent of points and the percent of GCs with a latency greater than 2^nx the average, where n = 1,2,3,...

Please note that these statistics were computed over the whole set of points (more than 1 million for each GC), as opposed to the charts which only used the highest 10000 points for readability. Also, we only increased n until the percentage of points became too close to 0. Tables 5, 6 and 7 illustrate the resulted statistics.

Table 6: Statistics regarding the average latency for READ and UPDATE operations for G1 GC

	READ	UPDATE
AVG(ms)	2.369	1.106
MAX(ms)	644.19	469.133
MIN(ms)	0.548	0.424
0.5x-1.5x AVG (%reqs)	95.325	99.029
0.5x-1.5x AVG (%GCs)	0.0	0.0
>2x AVG (%reqs)	1.73	0.766
>2x AVG (%GCs)	100.0	100.0
>4x AVG (%reqs)	1.107	0.526
>4x AVG (%GCs)	100.0	100.0
>8x AVG (%reqs)	0.603	0.124
>8x AVG (%GCs)	100.0	100.0
>16x AVG (%reqs)	0.596	0.005
>16x AVG (%GCs)	100.0	100.0

5. RELATED WORK

The performance of GCs has been an important topic ever since the managed runtime environments started to be heavily used both in research, as well as in production. The topic

Table 7: Statistics regarding the average latency for READ and UPDATE operations for ConcurrentMarkSweep GC

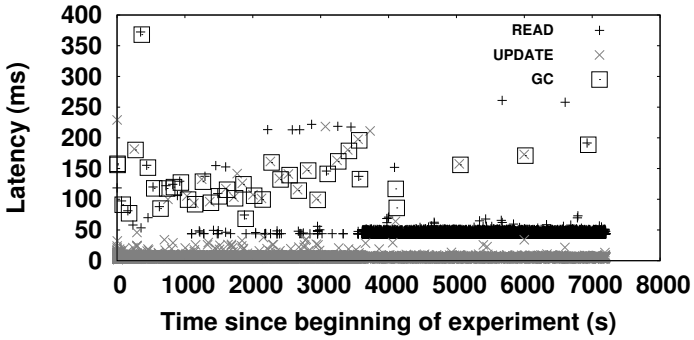
	READ	UPDATE
AVG(ms)	3.494	1.08
MAX(ms)	865.518	669.843
MIN(ms)	0.596	0.496
0.5x-1.5x AVG (%reqs)	53.382	98.811
0.5x-1.5x AVG (%GCs)	0.0	0.0
>2x AVG (%reqs)	3.828	0.921
>2x AVG (%GCs)	100.0	100.0
>4x AVG (%reqs)	2.983	0.556
>4x AVG (%GCs)	100.0	100.0
>8x AVG (%reqs)	2.958	0.066
>8x AVG (%GCs)	100.0	100.0
>16x AVG (%reqs)	0.003	0.005
>16x AVG (%GCs)	100.0	100.0

became hot once the GCs had to be adapted for multicore systems and the algorithms turned challenging. With the growth in the number of cores of today's computing machines, a simple serial collector is not suitable any more. However, because of the synchronization needed when moving objects, the current GCs are not fully concurrent either. Thus, lots of studies and improvements have been proposed in this area in the last few years.

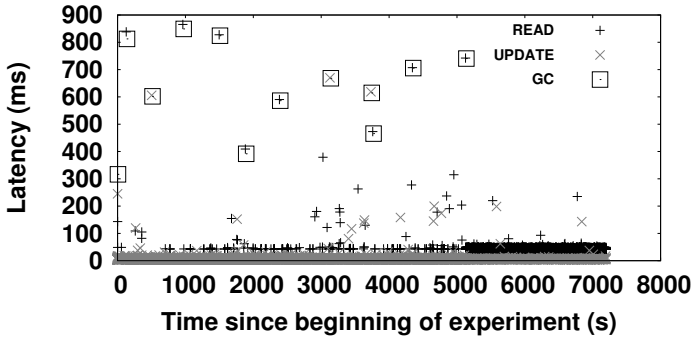
The Garbage First (G1) GC [9], evaluated in this paper, achieves real-time goals with high probability. It partitions the heap in fixed-size regions and any set of regions can be chosen for collection in order to ensure the time requirements. Besides G1, another two widely used JVM collectors have been evaluated in this work: ConcurrentMarkSweep [10] and ParallelOld [17].

Gidra et al [12] present in their paper a scalability study for these three main GCs in OpenJDK, i.e., ParallelOld, ConcurrentMarkSweep and G1. They evaluate the GCs on a 48-core NUMA machine, with the DaCapo benchmark suite. They point out that the current GCs do not scale with the number of threads and that the GC represents a significant bottleneck in multicore systems. They also identify *remote scanning* and *remote copying* as the most important problems concerning scalability. They add in [13] that another significant performance issue is the lack of NUMA-awareness when allocating objects. We consider that our contributions complement this work with results obtained by varying other properties than the application threads.

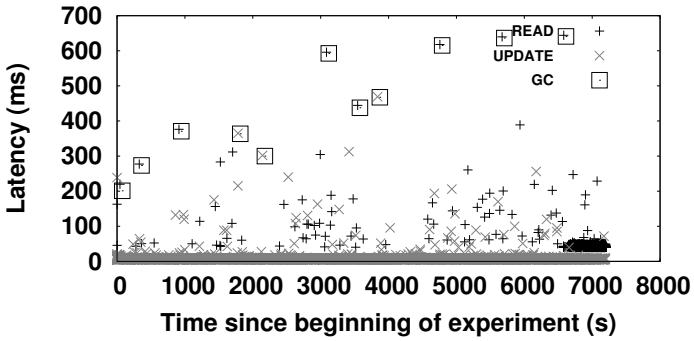
A new approach to developing concurrent garbage collectors employs Transactional Memory (TM). TM represents a technique that allows developers to write code without worrying about concurrent interleaving, by just embedding critical sections within "atomic blocks" (transactions). These blocks can then execute concurrently in isolation, possibly aborting upon conflict with the speculative execution of another atomic block. Software TM (STM) is not applicable to all problems from a pure performance perspective. However, hardware TM (HTM) appears to be the perfect solution for tackling specialized concurrency problems, such as fully con-



(a) ParallelOld



(b) CMS



(c) G1

Figure 5: Application response time for three GC strategies

current garbage collection. This new technique is used by Alistarh et al [1] for automating memory reclamation and reducing the pauses caused by the GC. They employ HTM in order for the GC threads to always have a consistent view of the variables accessed by other threads. The approach is implemented in C and adds automatic memory reclamation to classic data structures such as the non-blocking queue, linked list, skip-list and hash-table. However, even though StackTrack benefits of a superior performance as compared to other memory reclamation algorithms, it can also reduce the data structure throughput by up to 50%.

Another promising approach, described by Iyengar et al [14], is the Collie algorithm, the first one to propose HTM for Java VM. It is implemented on a specialized multicore CPU built by Azul Systems. While it represents an important first contribution in this field, the algorithm has two main drawbacks: it only uses a single thread for collection, which may be insufficient on large multicore systems, and it requires two passes over the object graph, increasing the probability of memory exhaustion during a collection. However, the main contribution is that this work indicates that the current problems of responsiveness and scalability can easily be addressed with the help of HTM.

6. CONCLUSIONS AND FUTURE WORK

In this work we evaluated the extent in which garbage collectors affect the execution of an application, with respect to throughput and responsiveness. We found that the results obtained from testing the GCs on a benchmark suite are contradicted to some degree by the real-life usage in a memory-intensive server application. We believe that one of the reasons for this was the small memory footprint of the benchmarks, which have been implemented in 2009 for smaller memories than on today's servers. However, by running the benchmarks we learned that the GCs are not always behaving as expected. Contrary to our assumptions, our experiments showed that:

- enabling the TLAB is not always beneficial, but can also decrease the performance of the application.
- the average pause time of the GC can increase with the young generation decrease, for the same heap size (this happens for ConcurrentMarkSweep and ParNew).
- for the benchmark suite, the default GC, ParallelOld, proved to be the most stable, having a very good performance in terms of pauses and total execution time.

However, our experiments with Apache Cassandra Server in combination with the YCSB client, using a large heap and Young Generation, indicated the opposite: in this situation, ParallelOld resulted in huge pause times for the application (up to 4 minutes long), becoming unacceptable in practice. Even G1 and ConcurrentMarkSweep collectors introduced pauses of a few seconds (up to 3.5 seconds for G1), which might affect the application in case of a real-time or distributed system, where the nodes need to communicate without delay. We showed that the garbage collections also harm the user experience: almost every peak in the client response time was associated to a collection on the server. Table 8 summarizes the benefits and disadvantages of the three main GCs, based on our set of experiments.

Table 8: Advantages and disadvantages of the three main GCs, according to our experiments

GC	Experiment	Throughput	Pause Time
ParallelOld	DaCapo	good	short
	Cassandra	good	unacceptable
CMS	DaCapo	fairly good	acceptable
	Cassandra	fairly good	significant
G1	DaCapo	bad	unacceptable
	Cassandra	fairly good	significant

Further on, we plan to implement and thoroughly test a garbage collector that uses HTM. The state of the art indicates that HTM could bring a lot of benefits for the current GC implementations and it could soon be found in any commodity hardware. We aim to repeat this evaluation of the GC impact on application execution and compare the new approach to the current available GCs.

7. REFERENCES

[1] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, 2014.

[2] Apache. The apache cassandra project. `http://cassandra.apache.org/`, 2014.

[3] 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[4] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 22–32, New York, NY, USA, 2008. ACM.

[5] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, 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. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[9] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM.

[10] D. Detlefs and T. Printezis. A generational mostly-concurrent garbage collector. Technical report, Mountain View, CA, USA, 2000.

[11] H. Gao, J. Groote, and W. Hesselink. Lock-free parallel and concurrent garbage collection by mark&sweep. *Science of Computer Programming*, 64(3):341 – 374, 2007.

[12] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. *Best papers from PLOS'11, SIGOPS Operating System Review (OSR)*, 45(3):15–19, 2011.

[13] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. A study of the scalability of stop-the-world garbage collectors on multicores. pages 229–240, 2013.

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

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

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

[17] Sun. Whitepaper: Memory management in the java hotspot virtual machine. `http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf`, 2006.

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