Performance Evaluation of Parallel Garbage Collectors in the Jalapeño Virtual Machine

Neha Mittal, Keshava P Subramanya, Chandra Krintz
University of California at Santa Barbara

{ nehamittal, keshava, ckrintz } @ cs.ucsb.edu

Abstract – As multi-core systems become ubiquitous, parallel computing is the new big thing today. As more and more software systems are designed to take advantage of this capability, the need for the JVM to support and efficiently handle Parallel Garbage collection becomes more relevant than ever before. In this paper, we present a detailed performance analysis of various Parallel Garbage Collection techniques in the Jalapeño Virtual Machine. We deployed and analyzed the merits and demerits of several popular garbage collection algorithms under single and multicore platforms.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**] Processors, Garbage Collection

C.4. [**Performance of Systems**] Performance attributes

General Terms

Parallel, Garbage Collection, Analysis, Design, Memory Management, JVM

Keywords

JikesRVM, Parallel Garbage Collection, Dual Core, Ubuntu

1. Introduction

Garbage collection (GC) is one of the most important and complex parts of the runtime system of programming languages. Parallel collectors take an orthogonal approach to the problem of reducing collection pause time. Rather than decreasing the total amount of work performed during a particular collection as generational collectors do, parallel collectors merely mitigate the effect of this work by running in parallel with the mutator (client). While a parallel collector still imposes some overhead cost on the system, it eliminates the long pause times associated with stop-the-world collection

We take four popular current day Garbage collection techniques namely – Generational Mark and Sweep (GenMS), Generational Copying (GenCopy), Copy Mark and Sweep (CopyMS) and Mark and Sweep (MS). IBM's JikesRVM installed on x86 with multi-core support forms our platform for deploying the above mentioned GC techniques. SPECJBB2005 was used as the server benchmark as load onto the JVM.

Our analysis revolves around three important parameters
- Throughput (in the context of SpecJBB2005 benchmark)
- Total GC time
- Mutator pause time

2. Experimental Setup

The tests were run almost exclusively on Ubuntu Linux 6.06 with **SMP enabled** on Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz. The total available physical memory was 1GB. The JikesRVM was used with various Garbage Collection Algorithms using the FastAdaptive Compiler. The latest SPECJBB2005 was the benchmark that was run. For all the experiments, we used a common workload of 8 warehouses which took four minutes per run.

3. Garbage Collectors in Java

This chapter presents a brief overview of the parallel garbage collectors that we studied and analyzed.

3.1 Mark and Sweep (MS)

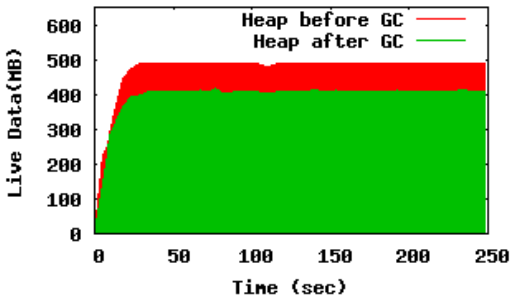

Figure 1: GC memory behavior for the Mark and Sweep

Mark-Sweep (MS) garbage collector traverses the entire object reachability graph. Each object is marked when it is scanned during the search, and unmarked objects are known to be garbage. Here is the algorithm.

```
for each root variable r
    mark (r);
sweep ();
```

Figure 1 shows the amount of memory live objects consumed following each GC, as well as the amount of memory freed when dead objects were reclaimed. The cost of this collector is proportional to the size of heap. Once the application ramps up, nearly equal memory is freed at each run of the GC.

3.2 Generational Mark and Sweep (GenMS)

It has been empirically observed that in many programs, the most recently created objects are also those most likely to quickly become unreachable (known as infant mortality or the generational hypothesis). A generational GC divides objects into generations and runs more frequently on the younger generations than on the older ones.

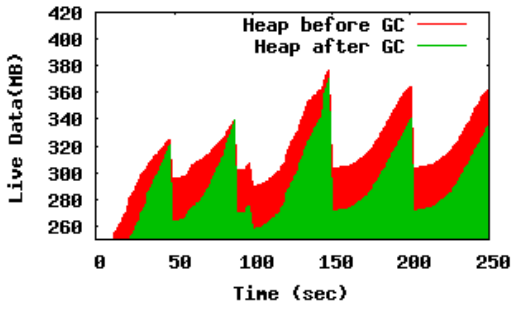

Figure 2: GC memory behavior for the Generational Mark and Sweep

Furthermore, the runtime system maintains knowledge of when references cross generations by observing the creation and overwriting of references (Remembered Set). When the garbage collector runs, it may be able to use this knowledge to prove that some objects in the nursery set are unreachable without having to traverse the entire reference tree. If the generational hypothesis holds, this results in much faster collection cycles while still reclaiming most unreachable objects. In a

generational Mark-Sweep Algorithm, the Mark-Sweep algorithm is used in both the nursery and the mature area. Since the collection runs on the whole heap occasionally and results in reclaiming large number of objects at once, the GC memory behavior is a Saw-tooth graph as shown in the figure-3.

3.3 Copy Mark and Sweep (CopyMS)

CopyMS uses two memory regions. New objects are allocated sequentially into the first region, which is a copying space. When the region is filled, reachable objects are copied into the second space, which is managed using Mark-Sweep. No write barrier is present, and every collection is performed over the whole heap

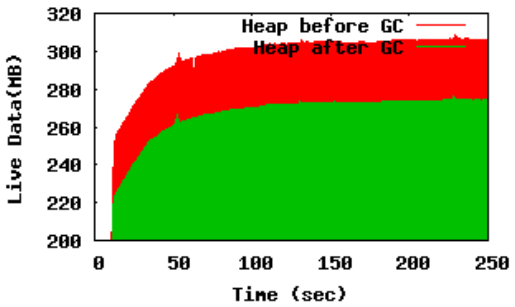

Figure 3: GC memory behavior for the Copy Mark and Sweep

Due to the collection being performed on the whole heap every time, the saw-tooth effect seen in the Generational GC techniques is not visible in the CopyMS.

3.4 Generational Copying (GenCopy)

GenCopy is a generational scheme in which both the mature and nursery space is managed with a standard Copy approach. Objects are allocated in the nursery until its current semispace is full.

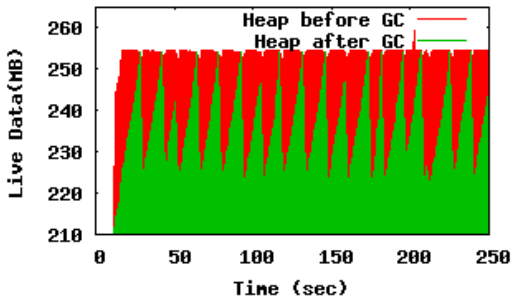

Figure 4: GC memory behavior for the Generational Copying

Then, only the nursery is collected, copying its live data into the other semispace. If an object survives long enough to be considered old, it can be copied out of the nursery and into the mature space. Eventually, the mature space will be filled up and the whole heap will be collected then. As the GC is invoked on the whole heap when a certain level threshold is reached, the used heap before very GC collection appears to be nearly a constant.

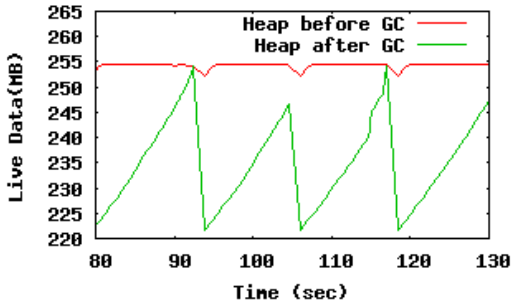

Figure 5: GC memory behavior for the Generational Copying (detailed)

Whole heap collection is done only if the nursery size falls below a static threshold. As this run was obtained on a 500MB total heap size, we can see that a collection is run when the static threshold (of nearly half the size of provided heap size ~ 250MB) is reached.

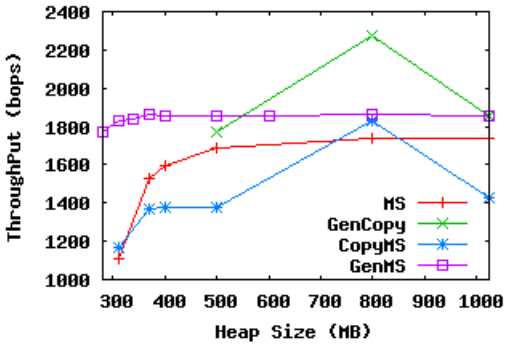

Figure 6: Throughput of MP GC techniques

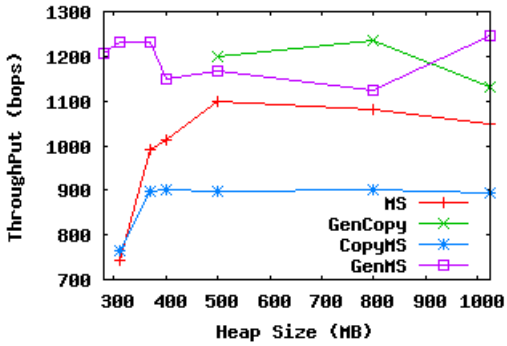

Figure 7: Throughput of SP GC techniques

4. Analysis of Parallel GC techniques

We now take a look at the following properties of the GC techniques on Single Processor (SP) and Multi Processor (MP) machines.

- Throughput: This metric is from the SPECJBB2005 benchmark and represents the number of operations per second (bops) that can be performed in a fixed amount of time, typically four minutes.

- Total GC time: Total time taken by the collector during Garbage Collection.

- Pause times: This represents the individual GC Collection times, during which the mutator is typically stopped.

4.1 Throughput Analysis

One quick observation one could make is about the throughput nearly doubling in the MP case. One could be tempted to attribute this to the Parallel Garbage collection. On the contrary, we find that it is one of the less important factors that contribute to this effect. The MultiCore version of the runs had availability of more computing power and access to true parallelism was nearly double that of the uniprocessor case.

GC	MS	GenCopy	CopyMS	GenMS
Growth	48%	68.9%	70.2%	53%

Table 1: Growth in throughput
(avgMP-avg SP) / avg SP

GC	MS	GenCopy	CopyMS	GenMS
Growth	54.8%	40.5%	45.06%	49.8%

Table 2: Growth in number of collections

One important observation to be made from Table 1 and Table 2 is that, thought the trends (relative ordering) of throughput of GC techniques are nearly the same, the difference in relative growths of each collector.

The analysis is further clouded by the fact that there are too many unknowns as the number of garbage collector runs also varies widely.

4.2 Total GC times

In our experience, the total GC time was nearly the same for both SP and MP runs. This could probably be accounted for the fixed time runs of the SPECJBB2005 benchmark. One striking

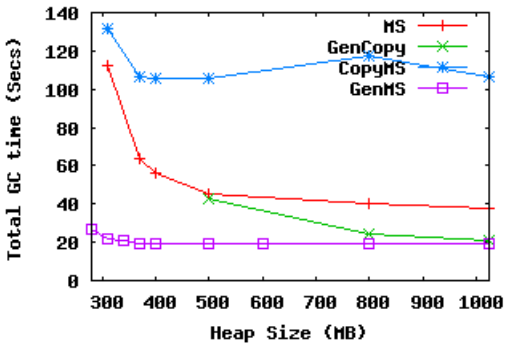

Figure 8: Total GC time in MP System

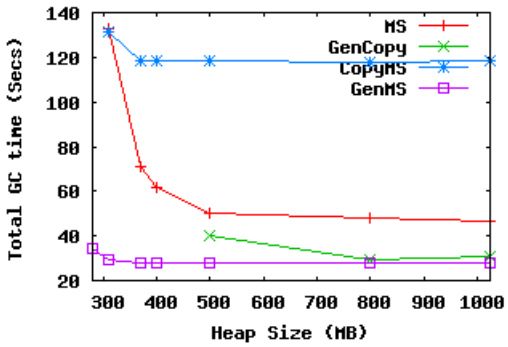

Figure 9: Total GC time in SP System

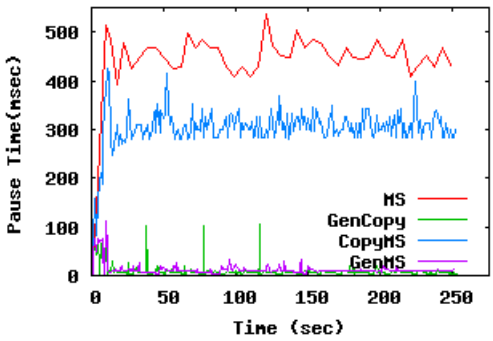

Figure 10: Pause time in MP System

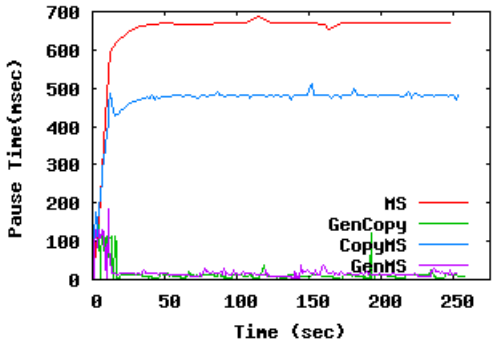

Figure11: Pause time in SP System

observation is that the total number of GC runs has increased, by nearly 50% (as in Table 2) in some cases, but the total GC time appears to be fixed. Figure 8 has the total GC time of the MP GC collectors.

4.3 Pause Time

We observe from Figure 10, 11 that the general trends are nearly the same for MP and SP systems, however, the graphs are smoother in the SP system. This could be accounted for the asynchronous Parallel GC collection that happens in the MP systems.

Therefore, we can say that parallel system can prove to be beneficial for the interactive applications, wherein the mutator is paused for less time and gives better throughout due to availability of parallel computing units.

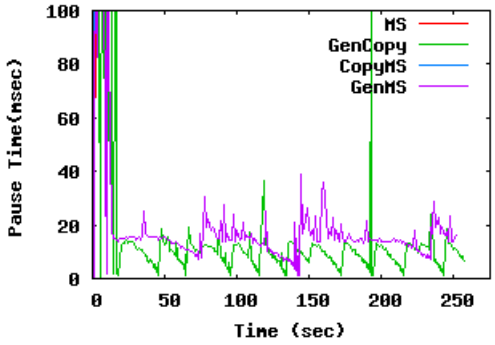

Figure12: Pause time in MP System for GenMS and GenCopy collectors(detailed)

Fig 12 shows the pause times for the GenMS and GenCopy collectors on the multiprocessors system. The GenCopy collector seems to have the smallest mutator pause time. We observed that the generational collector's pause time is much lesser than that of MS and

CopyMS. This could be accounted to its inherent design of performing frequent collection on small nursery size.

4.4 Summary of Observations

GenMS and GenCopy displayed some remarkable qualities. As seen from table 3, they had relatively low pause times and high throughput compared to MS and CopyMS. Under memory systems such as server environments, where most jobs are non interactive jobs, GenCopy may be the most suitable. However, GenCopy crashed under low memory conditions.

This could be attributed to the SemiSpace Strategy of Copying collectors which makes them unsuitable in low memory devices. GenMS, the real hero of the day, had nearly the same throughput as GenCopy, but had best, nearly constant total GC time and Pause time, even under low memory constraints. Unlike GenCopy, it had no spikes in the pause times and hence, works best even for realtime and interactive jobs.

5. Related work and Conclusion

A similar study of Parallel was conducted by Attanasio et al. In our work, we were able to verify some of the results in that paper. However, in some of the experiments, we have some conflicting results. This could be attributed to the fact that the experiments conducted on different benchmarks. As GC technique performance vary greatly based on mutator memory behavior.

Our observations can be summarized by the table below. Both GenMS and GenCopy work well. But GenCopy has issues with memory constrained systems and has a few but high pause times. This makes the GenMS the ideal choice of in case of interactive and real time applications

GC	Throughput	TotalGC	Pause	Overall
GenMS	good	best	best	Best
GenCopy	best	good	good	Good
MS	average	average	poor	Average
CopyMS	poor	poor	average	Poor

Table 3: Ranks of Parallel Garbage Collectors running SPECJBB2005

6. References

[1] PAUL R. WILSON "Uniprocessor Garbage Collection techniques In ACM computing surveys

[2] LUKE DYKSTRA, WITAWAS SRISA-AN, J. MORRIS CHAN "An Analysis of the Garbage Collection Performance in Sun's HotSpotTM Java Virtual Machine"

[3] B. ALPERN, C. R. ATTANASIO, J. J. BARTON, M. G. BURKE et al "The Jalapeno Virtual Machine"

[4] STEVE BLACKBURN ROBIN GARNER DANIEL FRAMPTON "MMTk: The Memory Management Toolkit"

[5] A very interesting article about GC http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html

[6] Tuning Garbage Collection with the 1.4.2 Java[tm] Virtual Machine http://java.sun.com/docs/hotspot/gc1.4.2/

[7] The GC page http://www.cs.kent.ac.uk/people/staff/rej/gc.html

[8] An article about tuning of GC http://www-128.ibm.com/developerworks/library/j-jtp01274.htm

Project Self Assessment

This project helped us get in depth understanding of Parallel Garbage Collection Strategies, their trade-offs, memory behaviors. Also, the project helped us understand the complex system of JikesRVM. We were able to contribute to the Ubuntu Dapper Community, some tips on how to run JikesRVM easily by enumerating the common pitfalls and issues based on our own experiences in the project.

Furthermore, this project helps us understand the value of discipline in project planning and execution which is evident from the shared document (googledocs).

For the above reasons, we strongly believe that we deserve the top grade for the project which is 1

Appendix

1. Enabling Multi-core support on Ubuntu - Dapper

* This should work for both Multicore systems and SMP systems

```
sudo apt-get install linux-686-smp
```

* This command will upgrade the kernel with SMP support. The default kernel does not support SMP or Multicores.

* You can check that this works by running the following and seeing two CPUs listed

```
cat /proc/cpuinfo
```

2. Enabling Virtual Machine Support in JikesRVM

Modifying file config/i686-pc-linux-gnu to add the following changes

```
RVM_FOR_SINGLE_VIRTUAL_PROCESSOR
to 0
```

Rebuilding RVM to incorporate these changes.

3. Compiling with the right version of Java

The 1.5 Version is required for compiling some parts of the JVM. Find out which version your /usr/bin/java is pointing to by

```
ls –l /usr/bin/java.
```

Follow the soft links to check if it is pointing to the right binary.

4. Running JikesRVM to use all processors

The RVM has to be invoked with

```
–X:processors=all
```

to instruct the RVM to use all the cores/processors in the system. If this option is not used, it will use the first available core/processor eben in SMP environment.

5. Checking to see if one GC thread is running per processor/core

The RVM has to be invoked with

```
-verbose:gc:8
```

The o/p should look something like this

```
SimplePhase.delegatePhase simple
[C] phase start-closure
SimplePhase.delegatePhase simple
[C] phase start-closure
  per-collector...
  per-collector...
Proc 1:     Working on GC in
parallel
Proc 2:     Working on GC in
parallel
```