On measuring garbage collection responsiveness

Tony Printezis

Sun Microsystems, MS UBUR02-311, 1 Network Drive, Burlington, MA 01803, USA

Received 29 April 2005; received in revised form 6 January 2006; accepted 24 February 2006
Available online 11 July 2006

Abstract

In this article we survey and evaluate methods for measuring and/or illustrating the responsiveness of low-latency garbage collectors. These methods include pause time distributions, minimum and bounded mutator utilization curves, percentile utilization curves, and cathedral graphs; the latter we introduce. We also discuss why we believe it is important to evaluate a garbage collector on its compliance against an application-specific goal. We propose to do so with two techniques: Vmetrics and GC overhead graphs.

© 2006 Sun Microsystems Inc. Published by Elsevier B.V. All rights reserved.

Keywords: Automatic memory management; Garbage collection; Garbage collection responsiveness; Vmetrics; Cathedral graphs; Percentile mutator utilization curves

1. Introduction

The recent popularity of languages such as the Java™ programming language and C# has finally established garbage collection, or GC, [12,19] as a valuable and advantageous memory management scheme. Now, even applications with stringent real-time responsiveness requirements are developed using garbage-collected languages. They range from very large applications that run on dedicated servers to very small ones that run on resource-constrained handheld devices. Thus, the research area on low-latency GCs has been very active recently, with new and improved techniques being proposed regularly.

Given this diversity of low-latency GC techniques, it is important to use well understood, widely accepted, and useful methods for comparing their responsiveness. In this article we survey comparison methods that have been proposed and used in the literature, and discuss their strengths and weaknesses. We also introduce a previously unpublished type of graph, percentile mutator utilization (PMU) curves, and we propose two new techniques: Vmetrics and cathedral graphs.

1.1. Running examples

Throughout this article we will illustrate our arguments by comparing metrics and graphs obtained from three GC traces. These traces were obtained by running the same application (the SPECjbb2000 benchmark [18]) on the same Java virtual machine and on the same eight-CPU server. But each trace was obtained by using a different GC
Throughout the paper, we will simply refer to these GCs as GC A, B, and C. The motivation for this is to encourage the reader to attempt to deduce how each GC operates, based only on the metrics and graphs we show, and without being prejudiced by their knowledge of how each GC algorithm is supposed to operate.

1.2. GC-centric vs. application-centric metrics

For most metrics and graphs shown in this paper, we had a choice to show them in either a GC-centric or an application-centric way. The former shows the behavior of the GC itself, the latter shows the behavior of the application, also referred to as the mutator, as affected by the GC. For example, a GC-centric metric would indicate that, in an application time period of 100 ms, the GC ran for 10 ms (we would refer to this as a 10% GC overhead on that time period). On the other hand, an application-centric metric would indicate that, in the same example, the mutator ran for 90 ms out of 100 ms (we would refer to this as a 90% mutator utilization on that time period).

Both approaches have been used in the literature, for example, pause time distributions (see Section 2) and Vmetrics (see Section 5) are GC-centric, while the MMU curves (see Section 3) are application-centric. The two approaches are mostly equivalent and we will not strongly recommend one over the other. However, we do find the GC-centric approach more intuitive for a few reasons. Since our goal is to illustrate the behavior of a GC, it is natural to focus on it, instead of obscuring it by using an application-centric metric. Additionally, GC-centric metrics can be more intuitively compared with pause time distributions (see Section 2), which are widely used and cannot be shown in an application-centric way.

Most importantly, during a time period with no GC activity, it is not necessarily the case that the application is actually making progress (it could be descheduled by the operating system, waiting for I/O, blocked for the results of an SQL query, and so on). However, during GC activity, it is guaranteed that the application is not running, even if the GC is descheduled by the operating system in favor of another process. Hence, focusing on when an application is not running, instead of when it may be running, is the more deterministic approach.

To be more specific, we consider the GC overhead on a single CPU to comprise the time intervals during which the application is suspended from running on that CPU in favor of the GC. During a stop-the-world pause, such intervals start as soon as all application threads are stopped and end when all application threads are restarted (we assume that the application threads start/stop atomically). During concurrent GC activity, such intervals are the ones during which a GC thread has been scheduled on the CPU by the operating system (Appendix A describes how we can approximate this in practice). We then consider the overall GC overhead on the application to be the GC overhead on all available CPUs divided by the number of all available CPUs.

In this article we illustrate all the metrics and graphs in a GC-centric way, except for all MMU-style curves, which have already been established as application-centric graphs.

1.3. Article overview

Section 2 covers pause time distributions and Section 3 describes minimum and bounded mutator utilization (MMU and BMU) curves. In Section 4 we discuss why we believe it is important to evaluate the responsiveness of a GC against an application-specific goal. We propose doing so using the Vmetrics in Section 5 and the GC overhead graphs in Section 7. A way to visualize the Vmetrics using spider graphs is given in Section 6. Section 8 introduces cathedral graphs, a combination of GC overhead graphs and MMU curves. Percentile mutator utilization and bounded percentile mutator utilization curves (PMU and BPMU), which are generalizations of MMU and BMU curves respectively, are introduced in Section 9. We conclude in Section 10. Finally, Appendix A describes a method for accounting concurrent GC overhead.

1 Even in the context of a multi-CPU environment, when a concurrent GC thread is running on a particular CPU it denies the application from running on that CPU.
2. Pause time distributions

In many publications that propose and evaluate new low-latency GC techniques, it has been an accepted practice to illustrate their responsiveness by looking only at the duration of their stop-the-world pauses. Typically, average and maximum pause times purport to illustrate the average- and worst-case scenarios, pause time histograms and sometimes standard deviation of pause times are also shown. Examples of such publications are the following [1, 13, 14, 16].

Figs. 1–3 show pause time histograms (using 5 ms “buckets”) for the three example GC traces we described in Section 1.1. The figures show some information on how the three GCs behaved: GC A seems to be the one that
imposed the longest (with several over 1 s) but also the smallest number of pauses, GC B is shown to have imposed shorter pauses on average than GC A, and GC C had the shortest maximum pause time (with all pauses being under 200 ms), but also caused the greatest number of pauses.

However, even though pause time distributions convey some important information on the behavior of a GC, they also hide some fundamental issues:

- **Pause proximity.** A GC that meets its pause time target of, say, 10 ms, but only allows the application to run for 1 ms between pauses, is not very attractive as it imposes a greater than 90% GC overhead on the application. It is now accepted that, in addition to statistics on pause time durations, it is also important to show a metric of how close together the GC pauses were clustered and, hence, to deduce how disruptive on the application the GC really was [6].

- **Pause scheduling.** We believe that it is also important to associate GC behavior with the application elapsed time, which a simple pause time distribution does not do. In this way, we could identify which phase of the application was disrupted by higher than desired GC activity and we would be in a position to associate this disruption with application-level events. For example, if the disruption only happens during the application’s initialization phase it might be acceptable to the user, as the GC parameters might have been optimized for the application’s steady-state phase. On the other hand, if the GC responsiveness worsens towards the end of the run, it might hint at a potentially serious problem (for example, fragmentation in the heap).

- **Concurrent processes.** Several GCs perform a significant fraction of their work concurrently with the application (concurrent marking, sweeping, and so on), in addition to stop-the-world pauses [8,13,16]. A pause time distribution does not reflect the amount of work the GC did concurrently and what portion of the application’s execution it affected. In certain situations, concurrent GC overhead can be considerable and very disruptive to an application’s responsiveness.

Despite these criticisms, keeping pause times short is obviously a very important goal for any low-latency GC. Pause time distributions are an intuitive and easy-to-understand illustration, even for users who are not experts in the GC/memory management area. Additionally, they are an important and necessary metric for work that attempts to decrease GC pause times or meet a given pause time goal.

3. MMU and BMU curves

To deal with some of the shortcomings of reporting pause times, which we described in Section 2, Cheng and Blelloch proposed the use of minimum mutator utilization, or MMU, curves to illustrate the responsiveness of a GC [6]. For an execution that starts at time $t_0$ and ends at time $t_e$, a time slice of duration $d$ is a time period $[t_1, t_2]$, where $t_1 \leq t_2$, $t_2 - t_1 = d$, $t_0 \leq t_2$ and $t_1 \leq t_e$ (notice that an execution has an infinite number of time slices of a particular duration). For a given time slice $T$ in an execution, mutator utilization is the percentage of $T$ that had no GC activity in it. For a given execution trace, the *minimum mutator utilization* for a time slice duration $d$ is the worst mutator utilization over all time slices of duration $d$ in the execution.

An MMU curve shows the minimum mutator utilization (on the y-axis) over a range of time slice durations (on the x-axis). Typically, the x-axis is in logarithmic scale to give more detail on the shape of the curve for small time slice durations. An example of an MMU curve is given in Fig. 4. It was generated from a synthetic trace which simulated 5 ms stop-the-world GC pauses interleaved with 5 ms of mutator work. The curve reaches values greater than 0 at the point on the x-axis which corresponds to the longest GC pause (in this case 5 ms, as all simulated pauses were 5 ms) and its y-value for the larger time slices gives an indication of the overall overhead the GC imposed on the application (in this case 50%). From our experience with MMU curves, oscillations on the curve, like the ones between $x = 0.01$ s and $x = 0.1$ s, denote regular interleaving of GC and mutator work at a regular rate (which is clearly the case in the synthetic trace). MMU curves have been widely used in the literature [2,5,6,10].

A bounded mutator utilization, or BMU, curve has a shape similar to that of the equivalent MMU curve. However, the y-value is not the MMU for time slice duration $d$ but, instead, the lowest MMU for all time slices greater than or equal to $d$. This eliminates the oscillations that MMU curves typically have and ensures that the BMU curve is monotonically increasing. Fig. 5 shows a BMU curve that was generated from the same synthetic trace as the MMU curve in Fig. 4. This BMU curve follows the bottom part of the corresponding MMU curve and the oscillations have
been replaced with a less visible step-like pattern. BMU curves were introduced by Blackburn et al. [4], but were christened by Sachindran et al. [17].

Figs. 6–8 show MMU curves for the three example GC traces and Figs. 9–11 show the equivalent BMU curves. The curves for GC A and B are quite unhelpful, as their shape is dominated by the longest GC pauses (almost 3 s and around 0.31 s respectively). They indicate, however, that the overall GC overhead of the two GCs was similar. GC C seems to have had higher overall GC overhead than the other two. However it also seems to have had better utilization at low time slice durations. The oscillations on its MMU curve indicate GC scheduling at a regular rate (this is also the case, to a lesser extent, in the case of GC A). Notice that this is less obvious in the BMU curves. Because of this, we tend to find MMU curves to be more useful.

MMU and BMU curves address the issue of pause proximity, for instance, how closely together pauses are scheduled (clusters of closely scheduled pauses would show up as low MMU). Additionally, when calculating them, we also can (and should) take into account any work the GC performed concurrently (see Appendix A). However, we still have some criticisms on these curves as comparison methods:

- **Worst-case metric.** MMU is an inherently worst-case metric (each point on the curve corresponds to the worst mutator utilization for the corresponding time slice duration). Such a metric can be useful for evaluating a hard real-time system, where the GC must be scheduled at regular and always predictable intervals [7]. As a result, the worst case should be close to the average case. However, there are a large number of systems with soft real-time responsiveness requirements. Such systems need to meet a given goal most of the time, but they can tolerate infrequent, and maybe large, deviations from it. When using MMU curves to evaluate such soft real-time GCs, their worst-case nature sometimes dominates and hides the average case, which is of more interest to the users of systems with soft real-time requirements.
- **GC activity scheduling.** The shape of an MMU curve does not illustrate how the GC’s behavior progressed during an application’s execution. For example, a GC trace ordered backwards would yield the same MMU curve as the original ordering. As we explained in Section 2, we believe this to be an important issue.

- **Non-intuitive.** It is important for developers of large commercial applications with soft real-time requirements, and their customers, to be able to evaluate different GCs from, say, different vendors. However, we do not feel that MMU/BMU curves are an intuitive and easy-to-understand illustration of the behavior of a GC. Application developers and users, who are not experts in the GC/memory management area, may not be able to appreciate the subtleties in the shape of the curves and their meaning.
4. The case for a soft real-time goal

There are large numbers of important applications that need to complete tasks (incoming requests usually) within a given, and typically short, amount of time, and need to do so with a high degree of predictability. Long and/or arbitrarily scheduled GC activity can disrupt the responsiveness of such applications. Hence, developers typically rely on low-latency GCs, which are now available in many commercial garbage-collected systems (for example, most current production Java virtual machines [3, 13, 15]), to minimize this disruption.

Applications fall within two categories: ones with hard real-time and ones with soft real-time responsiveness requirements. The former must always meet a given responsiveness goal; unexpected deviations from the goal would
have dire consequences (loss of life, financial loss, and so on) [7]. The latter can afford deviations from the goal, provided that they meet their responsiveness requirements most of the time.

In this article we mainly deal with applications with soft real-time responsiveness requirements. Examples of such applications are the following:

- **Telecommunication applications.** Several of our customers run applications that track or route telephone calls. A typical requirement of such an application is to respond within 500 ms of a call being initiated. Longer delays can be usually tolerated, provided that they do not happen frequently (for example, a single phone call that takes longer than usual to initiate will be quickly forgotten by a caller, but several of them in a day might cause the caller to defect to a different telephone company).

- **Video stream rendering.** Such an application needs to render frames at a fixed rate, for instance, 24 frames/s. Again, dropped frames can be tolerated (and may even go unnoticed by the viewer), provided that they do not happen frequently (as they might result in jerky and hence unwatchable video).

On the other hand, hard real-time applications are usually associated with safety critical systems. However, they are not exclusively limited to such systems. For example, cell phone manufacturers typically have hard real-time responsiveness requirements for the call-processing subsystem on their phones, even though it is not viewed as a safety critical application.

At a very high level, the applications we described above operate as follows. Incoming requests arrive and are inserted into a queue. As long as the queue is not empty, worker threads retrieve requests from the queue and process them. The processing time of a single request includes the time it takes for the request to get to the front of the queue and the time it takes the worker to process it. If a GC pause happens, the workers are “frozen” and are unable to process any requests. Meanwhile, the queue can grow (as it does not necessarily reside on the same process as the workers; instead, it could reside on a database, a separate load-balancing process, and so on). If the queue grows, so will the average processing time of a request. It follows that, in order for such an application to be able to process requests within the desired time, it also needs to ensure that the length of the queue stays within a certain bound. To achieve this, the GC needs to cooperate and ensure that (i) it does not stop the application for so long that the queue can grow and exceed the required bound and (ii) it allows the application enough time to “recover”, for instance, decrease the length of the queue, after a pause, before it stops the application again.

We have talked to many developers who work on this type of application. They have usually specified their GC behavior requirements as a single pause time target. This addresses point (i) above, but it does not address point (ii). A more accurate way to specify the desired behavior of a GC is by using a soft real-time goal, which has three components: (i) a maximum GC time, (ii) a time slice duration, and (iii) an acceptable failure rate. The soft real-time goal is interpreted thus:

\[
\text{In any interval of the application's execution of the given time slice duration, the GC should avoid using more than the allowed maximum GC time. Any violations of this goal should happen within the acceptable failure rate.}
\]

Notice that this interpretation is more general than the practice of some GCs (for example, the Metronome [2]), which is to schedule x ms of GC, followed by y ms of mutator work. Instead, we allow multiple GC pauses, or even concurrent GC activity, to take place during a time slice, as long as the goal is not violated. We denote a goal of x ms maximum GC time for every y ms time slice as \(x \text{ ms} | y \text{ ms}\).

We claim that such a soft real-time goal can be naturally deduced from the requirements of an application. Consider the video stream renderer mentioned above. It needs to make available a frame every 40 ms or so to meet the 24 frames/sec requirement. If it needs at most 30 ms to render a single frame, this yields a goal of 10 ms | 40 ms for this particular application. Further, if the maximum number of dropped frames that can be tolerated is, say, 10 per minute, then the acceptable failure rate of the GC not meeting the goal is around 0.7% (10 out of 1440 frames/min). Section 5 proposes a more precise format in which to express the acceptable failure rate.

We have so far demonstrated the need to drive a low-latency GC with a soft real-time goal. We also claim that, when we evaluate the responsiveness of the GC, it is also desirable to do so in terms of how well it meets this goal. Since

---

this goal has been deduced from the requirements of the application, it is not necessarily useful to the user to know whether the GC can meet either more or less stringent requirements. Consider again the video rendering example. It is irrelevant whether the GC meets a 1 ms | 40 ms or a 100 ms | 800 ms goal. In the former case, the renderer would not be able to take advantage of the additional available mutator time, as it would have to wait for 40 ms between making frames available. In the latter case, a 100 ms GC would prevent the rendering of three frames.

5. Vmetrics

To evaluate how well a GC meets a given soft real-time goal, as defined in Section 4, we have proposed a set of three metrics [8], which we refer to as the Vmetrics:

- \( V\% \) denotes the percentage of all time slices that are violating, (which are time slices whose GC time exceeds the maximum GC time of the goal).
- \( \text{avg}V\% \) denotes the average amount by which violating time slices exceed the maximum GC time, expressed as a percentage of the desired minimum mutator time in a time slice (which is the time slice duration minus the maximum GC time).
- \( wV\% \) denotes the excess GC time in the worst time slice, for instance, which is the excess GC time in the time slice(s) with the most GC time, expressed again as a percentage of the desired minimum application time in a time slice.

Informally, \( V\% \) shows what portion of the application’s execution was affected by GC activity that violated the desired goal, \( \text{avg}V\% \) shows the average extent of such violations, and \( wV\% \) shows the worst-case scenario.

Here, we give an example of how the \( \text{avg}V\% \) and \( wV\% \) metrics are calculated for a goal of 200 ms | 500 ms. For this goal, an average GC time of 305 ms over all violating time slices would yield an \( \text{avg}V\% \) metric of \( \frac{305 - 200}{500 - 200} = 35\% \) and a worst GC time of 410 ms would yield a \( wV\% \) metric of \( \frac{410 - 200}{500 - 200} = 70\% \).

The \( wV\% \) metric provides the same information as the MMU value, as defined in Section 3, for a given time slice duration; knowing one we can deduce the other. For example, a \( wV\% \) value of 50\% for a 200 ms | 500 ms goal translates to 150 ms of excess GC time in the worst time slice(s), which is the equivalent of 350 ms of GC time in the worst time slice(s), which yields a \( \frac{500 - 350}{500} = 30\% \) MMU.

Fig. 12 shows a table with the Vmetrics from the example GC traces, calculated against a generous 200 ms | 500 ms goal. It is obvious from the table that GC A’s responsiveness is the worst (which was expected, given the very long pauses it caused, illustrated in Fig. 1). GC B performed much better than GC A on average, but with a relatively high \( wV\% \) metric (due to the single maximum pause, illustrated in Fig. 2). Finally, GC C performed very predictably and never violated the goal.

It could be argued that bounding \( wV\% \) at 100\% obscures the worst case pause time, if it is longer than the time slice duration. For example, the worst pause time for GC A was 2.9 s (see Fig. 1), which would translate to a percentage much higher than 100\%. We could extend the definition of \( wV\% \) to use the longest time slice duration that contains no mutator activity, instead of the worst excess GC time as we defined it above. In the case of GC A, the extended \( wV\% \) would yield \( \frac{2907.9 - 200}{500 - 200} = 902.6\% \), but it would not change the \( wV\% \) values for GC B and C. However, any GC that has a \( wV\% = 100\% \), as we originally defined it, is bad in meeting the given goal; and knowing exactly how bad is not necessarily useful. A very long pause will not only cause \( wV\% \) to be 100\%, but would also increase \( V\% \) and \( \text{avg}V\% \) (for example, notice in Fig. 12 that GC A has very high \( V\% \) and \( \text{avg}V\% \) compared to GC B and C). Given this, our preference is to use the original definition of \( wV\% \) and we encourage the reader always to judge the behavior of a GC based on the values of all three metrics, not just one of them.

When evaluating a particular GC, it is important to do so against not only one but a variety of soft real-time goals. This way, we can show the range of goals for which the GC provides acceptable responsiveness and to discover what the limits are below which the GC is not useful any more (for instance, when the desirable time slice or GC overhead

<table>
<thead>
<tr>
<th>GC</th>
<th>( V% )</th>
<th>( \text{avg}V% )</th>
<th>( wV% )</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC A</td>
<td>7.78%</td>
<td>82.79%</td>
<td>100.00%</td>
</tr>
<tr>
<td>GC B</td>
<td>1.37%</td>
<td>10.91%</td>
<td>44.29%</td>
</tr>
<tr>
<td>GC C</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
</tbody>
</table>

Fig. 12. Vmetrics table for GC A, B, and C with goal 200 ms | 500 ms.
are too small for the GC to keep up with the application). Showing the Vmetrics for a variety of goals is a good way of doing so. However, we suggest that the goals against which a GC is evaluated should be those that were used as input for its operation, as we have done in our previous work [8].

Finally, we believe that the Vmetrics, either all three components or a subset of them, present an intuitive format in which to define the acceptable failure rate component of the soft real-time goal, as described in Section 4. For example, the user can set maximum limits for \(V\% = 2\%\) and \(wV\% = 10\%). Whether a GC can actually accept such an input and try to meet it is a discussion beyond the scope of this paper. We can, however, measure whether a GC can meet such a goal for a particular application. Fig. 12 shows that only GC C met the \(V\% = 2\%\) and \(wV\% = 10\%\) limits.

The Vmetrics provide a useful summary of a GC's compliance to a given soft real-time goal. We claim that the informal description we gave above is enough for an inexperienced user to understand what a set of Vmetrics means. Still, Vmetrics also have some drawbacks:

- **Soft real-time goal.** The Vmetrics can only be calculated against a given soft real-time goal. Nevertheless, as we described in Section 4, there is a natural goal for many applications that we are interested in. Additionally, several GCs actually use such a goal (for example, Garbage-First [8], Metronome [2]) to drive their behavior. Therefore, we do not think that this is a big disadvantage.

- **GC activity scheduling.** Second, like the pause time histograms (see Section 2) and MMU/BMU curves (see Section 3), Vmetrics do not show how the GC's behavior progressed during the application's execution.

6. Spider graphs

A useful way to visualize sets of Vmetrics, as described in Section 5, so that we can easily compare them, is to use spider graphs, also referred to as radar graphs or star graphs [11]. An example of such a spider graph is given in Fig. 13 (it is annotated more than those that follow, to help the reader understand it more easily). The graph has three axes, one for each metric. The axes are logarithmic. In our experience, most Vmetrics values are typically small (say 10%–15%) and using a logarithmic scale emphasizes the differences between them, while obscuring differences between large values. If any of the metrics gets very large, then we know that the GC has behaved quite badly; exactly how badly is not necessarily useful to know. So, using a logarithmic scale in this case is appropriate. The four concentric circles in the graph denote the 25%, 50%, 75%, and 100% limits. The three points on the axes that correspond to one set of Vmetrics are joined to form a shaded triangle. The actual values of the Vmetrics are explicitly given at the bottom.

Fig. 14 shows spider graphs generated from the data in Fig. 12. With a little practice, the reader can easily deduce which GC behaved better simply by comparing the size and shape of the triangles. In general, a smaller triangle denotes better compliance to the soft real-time goal. And the shape of the triangle indicates in which attribute(s) the GC did not perform well. We have found this representation of the Vmetrics to be less confusing than tables full of numbers (which we have admittedly used in past publications [8]).

An advantage of representing the Vmetrics as spider graphs is that we could easily show additional attributes. Instead of the triangles that represent three attributes (\(V\%, \text{avg}V\%,\) and \(wV\%\)), we can instead show polygons of higher degree to add additional attributes, such as application throughput, average and maximum pause times, overall GC overhead, and so on.

7. GC overhead graphs

In this section we show what we call GC overhead graphs which illustrate the compliance of a GC to a given soft real-time goal over time. Bacon et al. presented similar graphs when evaluating the Metronome [2] (they were the compliment to what we show here, though, as they showed mutator utilization and not GC overhead; both these approaches are broadly equivalent, however, as described in Section 1.2).

The \(x\)-axis in a GC overhead graph denotes the application elapsed time. The \(y\)-value at \(x = t\) is the time spent in GC in the time slice centered at \(t\). Only time slice durations equal to that in the soft real-time goal are considered. A horizontal line at \(y = \text{max. gc time}\) shows the maximum GC time set in the goal against which we are evaluating the GC.

Figs. 15–17 show GC overhead graphs for the three example GC traces. There are many observations we can make from them:
Fig. 13. Example of a Vmetrics spider graph.

Fig. 14. Vmetrics spider graphs for GC A, B, and C with goal 200 ms | 500 ms.

- The graphs show the compliance of each GC with the soft real-time goal very intuitively: whenever the graph moves over the horizontal line which denotes the maximum allowed GC overhead, the goal is violated. In this respect, GC A seems to have behaved the worst, with regular large violations. GC B did quite well overall, with only a few violations. Finally, GC C met the given soft real-time goal without any violations.
- The graphs give a clear representation of how the GC behaved over time and at which point of the execution violations took place. In the case of GC A violations happened regularly. Fig. 16 shows that GC B seems to have violated the goal mainly in the middle of the execution but then conforms to the goal for the rest of the execution.
- The graphs also give a good indication of the overall overhead that the GC imposed on the application (this can be deduced by observing what portion of the graph is black). In that respect, GC B seems to have imposed a greater overhead than GC A, and GC C a greater overhead than GC B.
- Another interesting observation from the three GC overhead graphs is how the three GCs scheduled their GC activity. GC A clearly scheduled all GC activity in regular stop-the-world pauses. The other two had regular stop-the-world pauses, but in addition they also had concurrent GC activity that is evident in the graphs. This is very clear in the case of GC B: notice the black bands that appear regularly at the bottom of the graph. The same pattern also appears in the graph for GC C, but it is less obvious, as concurrent activity happened with greater regularity.
- Comparing the Vmetrics values with the corresponding GC overhead graph, notice that the Vmetrics provide a summary of all the violations shown in the graph. So, we propose that they should also be included in each such graph, in the same way that we included the average and maximum pause times in the pause time histograms in Figs. 1–3.

From our experience with measuring the responsiveness of GCs, we have found the GC overhead graphs to be the most useful and intuitive of all the techniques we present in this article.

8. Cathedral graphs

The only potential drawback that the GC overhead graphs, presented in Section 7, have is that they must be drawn against a particular soft real-time goal. As we discussed in Section 5, we do not think this is a great disadvantage. Still,
it is desirable to show the same information as GC overhead graphs, but over a range of time slice durations. The fact that three metrics need to be represented on the same graph (application time, GC overhead, and time slice durations) naturally yields a 3D graph. However, detailed 3D graphs are usually difficult to read and understand when printed. Instead we propose a 2D graph with the third dimension being represented by color. We call these cathedral graphs, for reasons that will soon become apparent. We are not aware of any similar graphs proposed elsewhere.

In a cathedral graph, the $x$-axis represents application elapsed time, as in a GC overhead graph, and the $y$-axis represents a time slice duration (as in the MMU curves, we also use logarithmic scale for this axis). The color of every point $(x, y)$ on the graph, represents the percentage of GC that the time slice, centered at time $x$ and of duration $y$, has.
In this article, we have generated them in grayscale, since they print better this way. So, black represents 0% of GC, white represents 100% of GC, and we interpolate to deduce the shade of gray for other percentages. We could also generate them in color, and use different colors to indicate percentage ranges (for instance, black to blue represents 0% to 33%, blue to red represents 33% to 66%, and red to green represents 66% to 100%).

Figs. 21–23 show cathedral graphs for the three example GC traces. Figs. 18–20 include the parts of the graphs that correspond to the [80, 110] range in the x-axis to emphasize some of the subtle detail which is not clear on the other figures. It was their arch-like shape that prompted us to christen them “cathedral graphs.” From the figures we can observe several interesting patterns:
• From the graphs for \textbf{GC A} it is clear that GC pauses are represented with white lines that “fan out” towards the top of the graph. The longer the pause, the thicker the line and the larger the range of time slices on which it has a non-trivial effect. Interestingly, the graph in Fig. 21 shows very clearly how the overhead of two pauses overlaps as time slices get wider (observe how the “arches” of the long pauses that start around $x = 65$ s and $x = 95$ s meet at $x = 80$ s). This is an intuitive visual representation of why MMU curves do not monotonically increase.

• The graphs for \textbf{GC B} clearly show different behavior, compared to those for \textbf{GC A}. The pause times seem consistently short over time. However, notice that, for some periods of time (for example, [95, 105]), all time slices
have some level of GC overhead (for instance, the area is gray even for very small time slices). This represents concurrent GC activity that has a uniform effect over all time slices (as only one of the eight CPUs of the machine we ran on was dedicated to GC). Fig. 19 actually shows some thin black vertical lines in periods of concurrent GC activity (for example, two of them are obvious in the time interval [100, 105]). These lines correspond to periods of time when the concurrent GC thread was descheduled.

- The graphs for GC C show that this GC scheduled its activity much more frequently and that periods that had concurrent GC activity were shorter, but were scheduled closer together, than those in GC B (this is quite obvious when comparing Figs. 19 and 20). As a result, GC C imposed a higher GC overhead on the application, compared to the other two (the shades of gray at the higher time slices are generally brighter for GC C).

Cathedral graphs are a combination of MMU curves and GC overhead graphs; they provide strictly more information than either of them. An MMU graph summarizes every row of the corresponding cathedral graph by plotting the worst value for the row. The information shown on a GC overhead graph is the contents of a single row of the corresponding cathedral graph.

We do not claim that cathedral graphs are very intuitive, nor that they are particularly easy to understand by someone who is not a expert in the GC/memory management area. They do look better when viewed in color on a monitor, where one can zoom in and out and concentrate on specific sections of them. However, they show interesting aspects of the behavior of a GC and they are a useful addition to the other methods we have presented so far.

9. Percentile mutator utilization curves

We will conclude this survey with a style of graphs that has been discussed within the research community but that, as far as we know, has not been published elsewhere. They are a generalization of the MMU curves and we will refer to them as percentile mutator utilization, or PMU, curves. As we described in Section 3, the y-value of an MMU curve for time slice \( x = d \) represents the minimum mutator utilization that all time slices of duration \( d \) in the application’s execution have (that is, some time slices of duration \( d \) might have higher utilization than that but never worse). PMU curves are an attempt to generalize MMU curves and relax their worst-case characteristics. On a PMU curve for the \( N \)th percentile, the y-value at \( x = d \) shows the minimum utilization that \( N \% \) of time slices of duration \( d \) have, with the remaining \( (100 – N)\% \) of time slices of duration \( d \) having at least the same and maybe worse utilization. Naturally, a PMU curve of the 100th percentile is simply a standard MMU curve.

Figs. 24–26 show PMU curves for the three GC traces for a range of percentiles (85th, 90th, 95th, 97th, 98th, 99th, and 100th). From these three figures we can make several interesting observations:

- All PMU curves start from either a very high or a very low value. The reason for this is simple. For time slice durations smaller than the GC pause times, most time slices will fall either entirely inside or entirely outside a pause; hence they will have either 0% or 100% GC overhead. Interestingly, the lowest percentile curves for GC B and C do not start from 100% but from a lower number. This is due to the concurrent GC overhead that these two collectors impose outside GC pauses.
- From the figures, we can determine the variance of the GC overhead by looking at how far apart the curves for the different percentiles are. GC A is the worst in that respect, given that the curves for the lower and higher percentiles start converging after \( x = 5 \) s. This is no surprise; we have shown in previous sections that GC A regularly caused very long pauses. On the other hand, GC C seems to be the best in terms of predictability; all curves converge earlier than the other two GCs.
- Comparing Figs. 25 and 26 we can see that, even though the 100th percentile curve of GC B is below that of GC C up to \( x = 1 \) s, the lower percentile curves for GC B are actually higher than those of GC C throughout their entire non-zero range. This indicates that GC B generally had better responsiveness than GC C, apart from a few worse-than-normal pauses (which is clear when comparing the GC overhead graphs for the two GCs in Figs. 16 and 17). Standard MMU curves obscure this behavior because they show only worst-case performance. This is the main reason for our criticism of the MMU curves in Section 3.

Furthermore, just as BMU curves bound MMU curves, described in Section 3, we can also bound PMU curves. That is, the y-value at \( x = d \) is not the actual PMU value for time slice duration \( d \) but, instead, the lowest PMU value for all time slices greater than or equal to \( d \). We will refer to these curves as bounded percentile mutator utilization, or
BPMU, curves. Like BMU curves, and unlike MMU and PMU curves, this restriction ensures that BPMU curves are strictly non-decreasing. Figs. 27–29 show BPMU curves for the three GCs. The information shown on these curves is similar to that on the corresponding PMU curves, but without the oscillations of the latter.

PMU and BPMU curves have an interesting weakness. Curves for the higher percentiles might obscure the average behavior of the GC due to some brief periods of worse-than-normal behavior. On the other hand, curves for the lower percentiles might obscure this worse-than-normal behavior. Because of this, we strongly recommend that curves for several percentiles are shown together, as we did in Figs. 24–26, instead of being considered individually.
10. Conclusions

In this article we presented several methods that illustrate the responsiveness of low-latency GCs, we discussed their strengths and weaknesses, and we emphasized the importance of evaluating a GC against a goal specific to an application’s requirements. In our experience, the GC overhead graphs show information that has the best balance between being useful and easy to understand, even by non-experts. We also recommend the use of PMU curves for several percentiles in place of the standard MMU curves.

To help the reader understand why GC A, B, and C behaved in the way that we showed, we now reveal the algorithms behind them:
• **GC A**: a standard stop-the-world generational GC, with a parallel scavenging young generation and a mark-compact old generation [15].
• **GC B**: a mostly-concurrent generational GC, with a parallel scavenging young generation and a mostly concurrent mark-sweep old generation [15,16].
• **GC C**: the Garbage-First GC, a concurrent and parallel GC that schedules its GC activity in order to meet a given GC goal [8].

We ran GC C with a goal of 200 ms | 500 ms, which is the one we evaluated it against in Sections 5 and 7. We ran GC A and B with a young generation size that allowed the young generation collections to fall comfortably within that goal too (even when adding the concurrent GC overhead of GC B).

A method that we ignored throughout the article is to measure responsiveness at the application level, instead of at the GC level. For example, in the case of the video stream renderer we could measure the fraction of frames that were not rendered on time or, in the case of a telecommunications application, the percentage of phone calls that failed to initiate within 500 ms. Such a measure is clearly very important to the user of the application. However, it is not only the GC that can cause an application not to meet its requirements. Other such factors include synchronous JIT compilation, the operating system descheduling a mutator thread, a network problem preventing a request from being promptly delivered, and so on. So, an application-level metric is useful when the entire stack (hardware, operating system, virtual machine, and so on) is evaluated, for example, by a customer before a potential purchase. However, it should never be used on its own to evaluate the responsiveness of a GC, but only in conjunction with some of the methods we presented in this article.

**Acknowledgements**

The author is grateful to David Detlefs, Fabio Rojas, and Richard Jones for their constructive comments on several versions of this article and to Kim Perzel for her advice on the spider graphs. He would also like to thank Eliot Moss and David Bacon for their numerous suggestions and interesting discussions, and for bringing the percentile mutator utilization curves to his attention.

**Appendix A. Measuring concurrent garbage collection overhead**

In a previous publication [8], we took only stop-the-world pauses into account when calculating GC overhead. We accept that it is important to also consider concurrent GC activity when calculating GC overhead. In fact, we have done so for all the graphs and metrics that we surveyed in this article (MMU/BMU curves, Vmetrics, GC overhead graphs, cathedral graphs, and PMU/BPMU curves), with the exception of the pause time distributions.

If a GC has no concurrent activity, measuring its overhead on a single time slice, as required by most of the above methods, is straightforward. For example, to measure the GC overhead in the 200 ms time slice shown in Fig. A.1(a) we sum the duration of the pauses that are included in the time slice. There are two such pauses with a total duration of 75 ms. Therefore the GC overhead in this particular time slice is \( \frac{75}{200} = 37.5\% \) (the mutator utilization in that time slice is 62.5%).

Calculating the GC overhead in a time slice in the presence of concurrent GC activity is a straightforward extension to the calculation above. Consider the scenario in Fig. A.1(b). The 200 ms time slice shown in the figure contains two pauses, as well as a constant 10% concurrent GC activity. To calculate the GC overhead in this time slice we calculate the total duration of the GC pauses (in this case 50 ms) and multiply the remaining time slice duration by the concurrent overhead 10% × 150 ms = 15 ms, which yields a total of \( \frac{50 + 15}{200} = 32.5\% \) GC overhead for this time slice.

What remains is to show how to measure the overhead of concurrent GC activity. The technique we used is the following. At short intervals a thread that performs concurrent GC activity calculates \( t \), the elapsed time since the end of the previous interval, and \( v_t \), for how long the thread was scheduled since the end of the previous interval (in the Solaris operating environment we can measure the latter by using the `gethrvtime()` system call, which yields the time the calling thread spent scheduled since the start of the application). The ratio \( co = \frac{v_t}{t} \) yields the concurrent GC overhead of that thread on a single CPU during that particular interval. Dividing \( co \) further by the number of available CPUs yields the overhead of that thread on the application. We keep track of the start/finish time and \( co \) for all such intervals on memory buffers (one buffer per GC thread) and we dump the buffer contents at the end of execution. This
way, we can include these intervals in our GC overhead calculations in the way we described above. To treat all GC activity uniformly, we also represent each GC pause as such an interval, with a co of 100%.

Admittedly, some precision will be lost if the concurrent overhead varies substantially within the short time intervals that we measure. But, in our experience, concurrent GC overhead seems to be mostly uniform throughout an application’s execution (for example, when concurrent marking starts it imposes a uniform overhead until it completes; Fig. 16 shows a good example of this). Additionally, we can always adjust the target interval duration to get the best possible trade-off between accuracy and the number of intervals we need to store and process.

Appendix B. Trademarks

Java and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

References

