SCALABILITY OF GARBAGE COLLECTION IN JAVA-BASED DISCRETE-EVENT SIMULATORS

DAVID M. NICOL

Dartmouth College
Hanover, NH 03755
nicol@cs.dartmouth.edu

Keywords HotSpot, Java, memory, scalability

Abstract Discrete-event simulation tools based on the Java progamming language are finding increasing use. One of the attractions of Java is its automatic garbage collection capabilities. While garbage collection technology has markedly improved in recent years, the fact remains that when a simulation begins to stress the memory system, performance degrades. In this paper we develop a simple analytic model of a Java simulator's memory system behavior, as a function of problem size. We show that once the problem size causes the number of live objects to exceed 1/2 of the memory space allocated for them, that the frequency of garbage collection increases exponentially as the problem size increases. However, the problem size where this exponential growth is actually noticed may be significantly larger, provided that the native execution costs are large as compared to the cost of executing the the garbage collector. Our model predicts that Java based simulators will have performance that scales in problem size, up to a point where memory crisis suddenly degrades the performance dramatically. We show that this prediction is borne out in practice.

1 INTRODUCTION

Discrete-event simulation tools based on the Java progamming language are finding increasing use. Current examples include[17, 10, 6, 9, 3, 8, 5, 4, 1, 2, 7, 14]. Discrete-event simulation is just one application where Java is being used in ways that once only C++ was used. While similar in many ways, Java is distinctly different from C++ in that it provides built-in support for garbage collection. Both languages allow a program to dynamically request new objects from the runtime system. In C++ the programmer is responsible for returning an object once it is no longer needed. In Java that is not necessary. Its runtime system is able to determine automatically when an object is no longer being used by the program, and return it to the memory pool for reuse. As memory problems are one of the leading causes of bugs in C++ programs, automatic garbage collection is seen as a boon to programmer productivity.

In Java all objects are created on the runtime heap. Runtime memory management involves allocation of new objects from the heap; re-organization is triggered when a request is made that cannot immediately be honored. This re-organization involves compaction of objects that might still be in use, and reclamation of space used by objects that can be determined not to be in use.

One of the characteristics of discrete-event simulations that significantly affects performance is that simulations typically exhibit little locality of reference. Virtual memory does not buy much for a simulator—once the model state no longer fits in memory, the virtual memory system begins to thrash. The implication for Java-based simulators is that the heap should be sized so that it can reside entirely within the computer's physical memory, to forestall paging out to disk. This in turn means that the physical memory size constrains the size of the model that can be effectively simulated. Obviously, when the memory needs of the simulation approach the limits of available memory, the demands on the garbage collection system will be increased. This paper considers the question of *how large a model* can be simulated before these demands dominate, and *how quickly performance degrades* as a function of model size because of garbage collection.

We answer these questions using a simple analytic

model that describes the behavior of the HotSpot garbage collector[11] as a function of problem size. We develop a formula for the simulation's execution time per unit model that expresses the sum of the native execution costs associated with the object throughout its lifetime, the sum of an object's copying costs during "minor reclamation", and the average per object cost of a mark-and-compact garbage collection action[12], times the average number of times an object survives that collection. We see that the average number of times an object survives a mark-and-compact step is insignificant at small model sizes, but grows exponentially (in problem size) once the space needed by live objects exceeds 1/2 of the memory allocated for them. Our model predicts that a Java-based simulator's performance will scale with problem size until the cost of garbage collection (per unit simulation time) is of the same magnitude as the native problem execution time (per unit simulation), after which garbage collection costs dominate overall performance.

2 HOTSPOT GARBAGE COLLECTION

We model the actions of the Sun HotSpot garbage collector[11], found in JDK 1.3 and 1.4. This algorithm exploits the observation that many objects are short-lived and can quickly be identified as reclaimable, e.g., local variables that go out of scope after a procedure call. Such objects are reclaimed in a low-cost "Minor Reclamation" collection. A more traditional (and computationally expensive) full mark-and-compact "Full Compaction" phase[12] is invoked only when absolutely necessary. Details of this algorithm now follow.

The heap is partitioned into three sections named *Eden*, *Survivor*, and *Tenured*. Every request for a new object is satisfied from Eden, whose state is always comprised of a contiguous block of memory allocated to recently requested objects, and a contiguous block of free memory. A pointer demarks the division. A new object of size n bytes is given an n-byte block beginning at the pointer; and the pointer is advanced by n. A minor reclamation is triggered when an object is requested that cannot be satisfied in Eden.

The *Survivor* area is partitioned into two equal sized segments, to implement double buffering, one is empty. When a minor reclamation is triggered, the scavenge rules are applied to objects in Eden, and objects in the non-empty Survivor segment. Most objects that are not reclaimed are packed into the empty Survivor segment, and a copy-counter is incremented on each. However if an object has already been copied $K_T - 1$ times, it is copied instead to the *Tenured* segment.

The *Tenured* segment is organized like *Eden*, a pointer demarcates a region with allocated objects, and a region without. An object of size n bytes is copied beginning at the first address of the unallocated region, after which the pointer is incremented by n. A *Major Reclamation* is triggered when an object is to be copied into *Tenured*, but there is not enough space for it. A major reclamation implements a full mark-and-compact analysis of the entire heap. That is, the tree of live objects is traversed, all reachable objects are marked and compacted into a contiguous region in *Tenured*.

3 ANALYTIC MODEL

We now develop a simple analytic model of the costs of executing a Java-based simulation. We are interested in the simulator's behavior over a family of increasing large models. We abstract this notion by assuming that the simulation problem size can be parameterized by real-valued value n. For example, the system we later study is comprised of n concurrent TCP sessions that pass through a shared link in the Internet. In this model the bandwidth of the shared link increases in proportion to the number of TCP sessions, and as a result the simulation workload increases linearly with n. We denote the rate (in simulation time) at which new objects are created as $\lambda_o(n)$. We assume that an object is either short-lived (with lifetime L_s) or long-lived (with lifetime L_l). The fraction of short-lived objects is α_s, the fraction of long-lived objects is $\bar{\alpha_s} = 1 - \alpha_s$; the average object lifetime is thus $L_{avg} = \alpha_s L_s + \bar{\alpha_s} L_l$. A short-lived object requires an average of m_s bytes, a long-lived object requires an average of m_l bytes. The average object size is thus $m_{avg} = \alpha_s m_s + \bar{\alpha_s} m_l$. The average *space-time product* is $S_{avg} = \alpha_s m_s L_s + \bar{\alpha_s} m_l L_l$.

We let γ denote the average execution time per live byte per unit simulation time. We assume this cost is independent of the size of the model.

By Little's Law[13], the average number of bytes that are live at any instant in a simulation of problem size n is

$$\begin{aligned} L(n) &= \lambda_o(n) \cdot (\alpha_s m_s L_s + \bar{\alpha_s} m_l L_l) \\ &= \lambda_o(n) S_{avg}. \end{aligned}$$

Consequently the average execution time per unit simulation time for a problem of size n is $L(n)\gamma$.

Now consider the memory system. We suppose that the cost per byte of copying an object during a minor reclamation is β_m. We account for the cost of minor reclamations by considering how many times an object is copied, on average. We assume that long-lived objects always live long enough to be tenured, so that the total cost of minor reclamations on a long-lived object is $\beta_m m_l K_T$. A short-lived object may be copied fewer than K_T times however.

A minor reclamation is triggered once *Eden* is exhausted. If we denote by X_i the object size of the i^{th} new object since the last minor reclamation, and let N be the (random) number of objects needed to exhaust *Eden*, then

$$\sum_{i=1}^{N-1} X_i < M_E \leq \sum_{i=1}^{N} X_i.$$

N is what is known as a "stopping time" [16], which implies that $E[\sum_{i=1}^{N} X_i] = E[N]E[X_i]$. Applied to the inequality above, this tells us that $E[N] \approx M_E/m_{avg}$. Then, since new objects are created at rate $\lambda_o(n)$ it follows that the average number of simulation time units between minor reclamations is (approximately) $M_E/(m_{avg}\lambda_o(n))$. It follows that the number of minor reclamations that occur in a short-lived object's lifetime is $\lambda_o(n)L_s m_{avg}/M_E$ when this expression is less than K_T, and is K_T otherwise.

We can approximate the average rate of copying costs due to minor reclamation by associating with an object the average cost we expect it to incur, at the point of its creation. This gives rise to a minor reclamation cost rate rate

$$\begin{aligned} C_m(n) &= \beta_m \lambda_o(n) \Bigg(\alpha_s m_s \times \\ &\quad \min\{\frac{\lambda_o(n) L_s \cdot m_{avg}}{M_E}, K_T\} + \bar{\alpha_s} m_l K_T \Bigg). \end{aligned} \tag{1}$$

From this we see that the copying cost of a short-lived object grows in the square of the object creation rate, until $\lambda_o(n)$ reaches a threshold

$$\lambda_t = \frac{K_T M_E}{L_s m_{avg}}$$

after which short-lived objects are copied only K_T times and are then tenured.

Next we turn to major reclamations. A full garbage collection is triggered either when *Tenured* is exhausted, or when a minor reclamation fills a survivor buffer. Java garbage collectors are tuned to avoid the latter occurence, and so we will not model that effect. After a full garbage collection step, *Tenured* will contain all the objects that are alive at the instant of collection. Little's Law tells us that the average number of short-lived objects alive at any instant is $\alpha_s \lambda_o(n) L_s$ and that the average number of long-lived objects at any instant is $\bar{\alpha_s}\lambda_o(n)L_l$. It follows that the average amount of *Tenured* memory occupied by live objects after a major reclamation (what we will call the problem's *kernel*) is

$$\begin{aligned} k(n) &= \lambda_o(n)(\alpha_s L_s m_s + \bar{\alpha_s} L_l m_l) \\ &= \lambda_o(n) S_{avg}. \end{aligned}$$

This leaves $M_T - k(n)$ bytes available to receive tenured objects, before another major reclamation is triggered.

Let $\Lambda_T(n)$ denote the average rate at which objects are copied into *Tenured* while running a problem of size n. When $\lambda_o(n) < \lambda_t$ all short-lived objects are reclaimed during minor reclamation. When $\lambda_o(n) \geq \lambda_t$ the short-lived objects become tenured also. In summary then

$$\Lambda_T(n) = \begin{cases} \bar{\alpha_s}\lambda_o(n) & \text{for } \lambda_o(n) < \lambda_t \\ \lambda_o(n) & \text{for } \lambda_o(n) \geq \lambda_t \end{cases}$$

We again use stopping time arguments to estimate how often a full reclamation is triggered. The amount of free space in *Tenured* after a full reclamation is a random variable, say F, with $E[F] = M_T - k(n)$. The i^{th} object tenured has (random) size X_i. If N is the smallest number of objects copied so that $\sum_{i=1}^{N} X_i \geq F$, then N is a stopping time and

$$\begin{aligned} E[F] = M_T - k(n) &\geq E[\sum_{i=1}^{N} X_i] \\ &= E[N]E[X] \end{aligned}$$

from which we estimate that

$$E[N] \approx \begin{cases} \frac{M_T - k(n)}{m_l} & \text{for } \lambda_o(n) < \lambda_t \\ \frac{M_T - k(n)}{m_{avg}} & \text{for } \lambda_o(n) \geq \lambda_t \end{cases}$$

The average amount of simulation time between successive full reclamations is approximately $E[N]/\Lambda(n)$. We model the cost of a full reclamation as being proportional to the number of live bytes, which gives rise to an average cost of $\beta_f k(n)$, for some per-byte cost β_f. The cost rate per unit simulation time of full reclamation is thus

$$C_f(n) = \frac{\beta_f k(n)}{E[N]/\Lambda(n)} \tag{2}$$

$$= \begin{cases} \lambda_o(n)\left(\beta_f \bar{\alpha_s} m_l \frac{k(n)}{M_T - k(n)}\right) & \lambda_o(n) < \lambda_t \\ \lambda_o(n)\left(\beta_f m_{avg} \frac{k(n)}{M_T - k(n)}\right) & \lambda_o(n) \geq \lambda_t \end{cases}$$

Our model explains that the overall execution cost per unit simulation time is the sum $C(n) = \gamma L(n) + C_m(n) + C_f(n)$. From equations (2) and (3) we have a full equation for $C(n)$, which is expressed in Figure 1.

We can interpret this last expression for $C(n)$ as the object creation rate times the sum of average per-object costs of execution, minor reclamation, and major reclamation. Because the units of γ are execution time per byte per unit simulation time, γ times the average object space-time product S_{avg} gives the object's execution cost. The units of β_m are execution time per byte, so β_m times the average number of times an object's bytes are copied gives the minor reclamation cost. Likewise, the units of β_f are execution time per byte, so the product of β_f times the average tenured object size times the number of times a tenured object is in *Tenured* when a major reclamation occurs gives the average per-object reclamation cost.

The interesting term in Equation (3) is $F(n) = k(n)/(M_T - k(n))$, which is the average number of times an object survives the full reclamation phase. $F(n)$ grows very rapidly when $k(n)$ approaches value M_T. To see this, let n_1 be the problem size at which $k(n_1) = M_T/2$, n_2 be the problem size at which $k(n_2) = k(n_1) + (M_T - k(n_1))/2$, and in general, n_i be the problem size at which $k(n_i) = k(n_{i-1}) + (M_T - k(n_{i-1}))/2$, i.e. $k(n_i)$ is larger than $k(n_{i-1})$ by 1/2 of the *Tenured* region unoccupied by the kernel for problem size n_{i-1}. Working through the math we get

$$\begin{aligned} F(n_{i+1}) &= \frac{k(n_{i+1})}{M_T - k(n_{i+1})} \\ &= \frac{k(n_i) + (M_T - k(n_i))/2}{M_T - (k(n_i) + (M_T - k(n_i))/2)} \\ &= \frac{M_T}{M_T - k(n_i)} + \frac{k(n_i)}{M_T - k(n_i)} \\ &> 2F(n_i), \end{aligned}$$

the inequality following from the observation that $M_T > k(n)$. Thus we see that by increasing the kernel by only 1/2 of what is possible, then the average number of major reclamations an object undergoes more than doubles. Once the live objects occupy more than 1/2 of *Tenured*, the frequency of major garbage collection begins to increase exponentially. This tells us to expect exponential growth in the major reclamation costs. However, the onset of the problem size where this growth is actually noticed depends on application characteristics, as we will see.

It is instructive to consider how changes in the simulation model affects these costs. It is notationally clear that increases in problem size increase $F(n)$. Intuition tells us that increasing the average object size will increase the demand on memory. Perhaps more subtle are changes that increase object lifetime. For example, in a network simulation lifetimes of packet objects increase with the latency on links. Recall that $k(n) = \lambda_o(n) S_{avg}$; increasing object lifetime increases object space-time product, and so increases $k(n)$.

Given the threat of exponential growth in garbage collection costs, it is important to understand how much of the memory can be occupied by the kernel objects before this growth begins to dominate the overall execution costs. To get some insight into this, we suppose that minor reclamation costs are insignificant relative to other costs. Denote the ratio of an object's native execution cost to the cost associated with it on one pass through the mark-and-compact algorithm by B, e.g., for the case when $\lambda_o(n) \geq \lambda_t$ let $B = \gamma S_{avg}/(\beta_f m_{avg})$. Then suppose that a given user believes that garbage collection costs are excessive when they are X percent of overall running time or larger, i.e. that $\beta_f m_{avg} F(n) = X(\gamma S_{avg} + \beta_f m_{avg} F(n))$; equivalently, that $F(n) = BX/(1 - X)$. We ask how large the kernel is at this balance point. Writing $k(n) = \alpha M_T$, the solution for α in the equation $F(n) = BX/(1 - X)$ is $\alpha = BX/(1 + (B - 1)X)$. For example then if an object's native execution costs are 20 times larger than the cost attributed to it in one application of the mark-and-compact algorithm, and a user's tolerance for garbage collection overhead limits it to be 20% of the overall cost, then $XB = 20 \times .2 = 4$, making $\alpha = 4/4.8 = 0.83$. In this case we see that (just as intuition suggests) significant native workload offsets garbage collection costs, allowing 83% of *Tenured* to be filled with live objects and still maintain a relatively low garbage collection overhead. However, let the problem size grow so that 96% of the memory holds live objects, and the garbage collection costs grow by at least a factor of 4, and the overall execution time doubles. After this point very small changes in problem size exacerbate garbage collection markedly. This analysis suggests then that when there is a significant amount of intrinsic computational work per object, that garbage collection costs do not dominate until the *Tenured* area is nearly full, at which point they overwhelm performance.

$$
\begin{aligned}
C(n) &= \begin{cases} \gamma L(n) + \beta_m \lambda_o(n)(\alpha_s \lambda_o(n) m_s \frac{L_s \cdot m_{avg}}{M_E} + \bar{\alpha_s} m_l K_T) + \beta_f \lambda_o(n) \bar{\alpha_s} m_l \frac{k(n)}{M_T - k(n)} & \text{for } \lambda_o(n) < \lambda_t \\ \gamma L(n) + \beta_m \lambda_o(n) m_{avg} K_T + \beta_f \lambda_o(n) m_{avg} \frac{k(n)}{M_T - k(n)} & \text{for } \lambda_o(n) \geq \lambda_t \end{cases} \\
&= \begin{cases} \lambda_o(n) \left(\gamma S_{avg} + \beta_m (\alpha_s \lambda_o(n) m_s \frac{L_s \cdot m_{avg}}{M_E} + \bar{\alpha_s} m_l K_T) + \beta_f \bar{\alpha_s} m_l \frac{k(n)}{M_T - k(n)} \right) & \text{for } \lambda_o(n) < \lambda_t \\ \lambda_o(n) \left(\gamma S_{avg} + \beta_m m_{avg} K_T + \beta_f m_{avg} \frac{k(n)}{M_T - k(n)} \right) & \text{for } \lambda_o(n) \geq \lambda_t \end{cases}
\end{aligned} \tag{3}
$$

Figure 1: Full equation for $C(n)$

4 EMPIRICAL VALIDATION

We confirm the qualitative model predictions by examining the performance of the **SSFNet** network simulator[17]. The model we study is a classic one, of concurrent TCP sessions that share a link; the performance of various simulators on this model was analyzed in [15]. On one side of the link are TCP clients, on the other side are servers. Each client opens a TCP connection with its own server, which then transfers a long file. We increase problem size and workload by simultaneously increasing the number of TCP sessions, and the bandwidth of the shared link. We point out in [15], it is necessary to increase the bandwidth if we are to increase the execution workload of the model by increasing the number of concurrent TCP sessions.

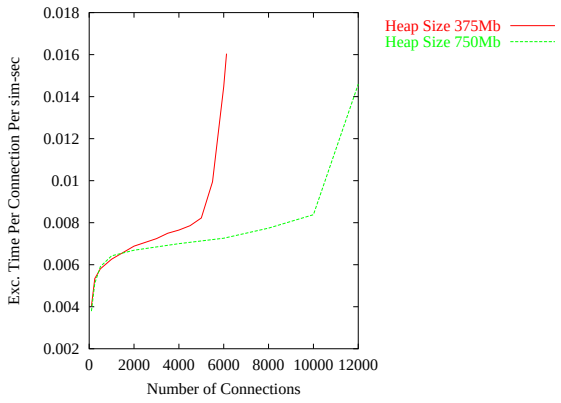

Figure 2: Performance of SSFNet as the Problem Size Grows

Figure 2) plots the average execution time per unit model size (connection) per simulation second of the **SSFNet** simulator[17]. Two curves are shown, one when the heap size is 375Mb, the other when it is twice as large. Both exhibit the exponential explosion our model explains. The slight growth in normalized execution time on smaller problem sizes might be explained by factors such as priority queue event lists, whose costs increase logarithmically in problem size. We have not yet instrumented this code to determine (for example) the comparative costs of native execution and garbage collection. Nevertheless this figure (and others like it) give us confidence that our model *explains* Java garbage collection costs well.

5 CONCLUSIONS

We consider the question of whether garbage collection in Java-based discrete-event simulators scales, in the sense of maintaining good performance as the problem size increases. Using a simple analytic model we answer the question affirmatively, with the caveat that as the problem size reaches a critical region its contribution to overall execution time increases very rapidly, ultimately dominating performance. These results confirm the utility of Java as a tool for discrete-event simulation, but warn that users need to understand the memory needs of their simulators and be sure that the platforms on which they run their simulations can meet the application's memory demands.

ACKNOWLEDGEMENTS

This research was supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, Dept. of Justice contract 2000-CX-K001, and Department of Energy contract DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

BIOGRAPHY

David M. Nicol is Professor of Computer Science at Dartmouth College, and Director of Research and Development at the Institute for Security Technology Studies. He earned a B.A. in mathematics at Carleton College in 1979, and M.S. and Ph.D. degrees in computer science from the University of Virginia in 1985. His research interests include high performance computing, performance analysis, simulation and modeling, and network security. He is co-author of the widely used text, *Discrete Event System Simulation, $3rd$ Edition* (by Banks, Carlson, Nelson, Nicol). He has been Editor-in-Chief of ACM Transactions on Modeling and Computer Simulation since 1997. He is a Fellow of the IEEE.

References

[1] Using Java for discrete event simulation. In *Proceedings Twelfth UK Computer and Telecommunications Performance Engineering Workshop*, pages 219–228, Univ. of Edinburgh, 1996.

[2] SPaDES/Java: Object-oriented parallel discrete-event simulation. In 35^{th} *Annual Simulation Symposium*, San Diego, April 2002.

[3] `http://javasim.ncl.ac.uk/`.

[4] `http://picolibre.enst-bretagne.fr/projects/flan/`.

[5] `http://ptolemy.eecs.berkeley.edu/ptolemyII`.

[6] `http://repast.sourceforge.net/`.

[7] `http://spe.univ-corse.fr/filippiweb/appli/debut.htm`.

[8] `http://www.javasim.com/`.

[9] `http://www.neosim.org/`.

[10] `http://www.threadtec.com/`.

[11] `java.sun.com/docs/hotspot/gc/`.

[12] Richard Jones. *Garbage Collection : Algorithms for Dynamic Memory Management*. John Wiley & Sons, Ltd., 1996.

[13] L. Kleinrock. *Queueing Systems, volume 1:Theory*. John Wiley and Sons, New York, 1975.

[14] P. L'Ecuyer, L. Meliani, and J. Vaucher. SSJ: A framework for stochastic simulation in Java.

[15] D. M. Nicol. Scalability of network simulators revisited. In *Proceedings of the 2003 Conference on Networked and Distributed Systems (CNDS)*, Orlando, FL, January 2003. To appear.

[16] H.S. Ross. *Stochastic Processes*. Wiley, New York, 1983.

[17] `www.ssfnet.org`.