Assessing the Scalability of Garbage Collectors on Many Cores

Lokesh Gidra[†] Gaël Thomas[†] Julien Sopena[†] Marc Shapiro[‡]
Regal-LIP6/INRIA
Université Pierre et Marie Curie, 4 place Jussieu, Paris, France
[†]firstname.lastname@lip6.fr [‡]marc.shapiro@acm.org

ABSTRACT

Managed Runtime Environments (MRE) are increasingly used for application servers that use large multi-core hardware. We find that the garbage collector is critical for overall performance in this setting. We explore the costs and scalability of the garbage collectors on a contemporary 48-core multiprocessor machine. We present experimental evaluation of the parallel and concurrent garbage collectors present in OpenJDK, a widely-used Java virtual machine. We show that garbage collection represents a substantial amount of an application's execution time, and does not scale well as the number of cores increases. We attempt to identify some critical scalability bottlenecks for garbage collectors.

1. INTRODUCTION

Managed languages such as Java and C# support fast and safe development of applications. With the improvement in techniques like Just In Time (JIT) compilation and automatic garbage collection, Managed Runtime Environments (MREs) become relevant for server applications such as JBoss, Tomcat or Oracle WebLogic Server. Therefore, the efficiency of MREs has become essential.

One of the attractive features of MREs is automatic garbage collection (GC). GC supports the MRE in ensuring type safety, and frees the programmer from the error-prone task of manually dealing with memory deallocation and heap organization. However, a substantial proportion of the resource requirement of MREs is due to GC. Hence, GC performance plays a vital role in MRE's performance, and therefore in application performance as well.

Contemporary many-core architectures provide more and more parallelism. To exploit this parallelism, a garbage collector typically uses multiple GC threads in order to reduce collection time[1]. With parallel garbage collection, stop-the-world pause time[2] is expected to be inversely proportional to the number of cores. We call this *GC scalability.*

Most current parallel garbage collectors are designed for Symmetric Multi-Processors (SMP) with a small number of processors. Furthermore, previous scalability evaluations of garbage collectors used relatively small multiprocessors [5, 15, 16]. Therefore, we believe it is time to re-evaluate the scalability of the current garbage collectors on contemporary hardware architectures, to verify whether the expected behavior is observed or not. In this work, we seek to answer the following questions:

- Is GC performance critical for the application? Otherwise, it is not worthwhile improving it.
- Does GC scale with the increasing number of CPUs? In other words, as expected by any parallel GC, does the stop-the-world pause time reduce as we add more and more cores?
- If the previous answer is "no", what are the bottlenecks affecting its scalability?

We experiment with the OpenJDK Java Virtual Machine because it is one of the most widely used JVMs. All the garbage collectors implemented in OpenJDK are generational. There are mainly four garbage collection techniques with several possible combinations of young and old generation collectors. We evaluate three combinations: Parallel Scavenge and Concurrent Mark Sweep with parallel young generation collection [8], and Garbage First [5]. All three are parallel and/or concurrent, hence are appropriate candidates for this evaluation. For more details on these GC algorithms, please refere to Section 4. We use the DaCapo benchmark suite [2] as it is representative of real-world Java applications. We considered all the benchmarks in the suite, and focused on three among the most scalable ones. Our evaluation platform is a 48-core NUMA machine.

The contributions of our paper are as follows:

- **DaCapo Scalability Test:** We evaluate the scalability of DaCapo benchmarks as a function of the number of application threads. *Sunflow*, *Lusearch* and *Tomcat* are among the most scalable benchmarks. Therefore, we focus on these three benchmarks for our tests.
- **GC Effect on Application:** We evaluate application scalability as a function of the number of cores (the

This research is supported in part by ANR projects Prose (ANR-09-VERS-007-02) and ConcoRDanT (ANR-10-BLAN 0208).

[1]The proportion of total execution time where the GC threads are active.

[2]When all the application threads are paused during the collection.

number of application threads and GC threads being equal to the number of cores). The experiment shows that the total execution time doesn't decrease with increasing number of cores. Furthermore, the proportion of pause time to total execution time increases. If GC pause time could scale, the application would scale too. Therefore, GC is critical to application scalability.

- **GC Scalability:** We evaluate GC scalability as a function of the number of GC threads, with constant number of cores and application threads, i.e. 48. Stop-the-world pause time measurements show that after a while, the pause time monotonically increases. The three GCs which we evaluated do not scale well as core count increases.

- **Factors Affecting GC Scalability:** We gathered object locality information. The numbers suggest *Remote Scanning* and *Remote Copying* of objects as the most crucial factors affecting scalability. Remote scanning refers to GC accessing live objects allocated on some remote memory node. Remote copying refers to copying live objects to a remote memory node. We also measure the effect of GC *Load Balancing*; disabling it improves performance in some cases, with no degradation in others. The results also show that the load is usually unbalanced after root set collection. Therefore, a scalable load balancing scheme is indeed required.

The remainder of the paper is structured as follows. Section 2 describes the experimental results. Section 3 identifies possible factors causing this behavior. Section 4 compares with related work. Finally, Section 5 concludes with some directions for improvement and future work.

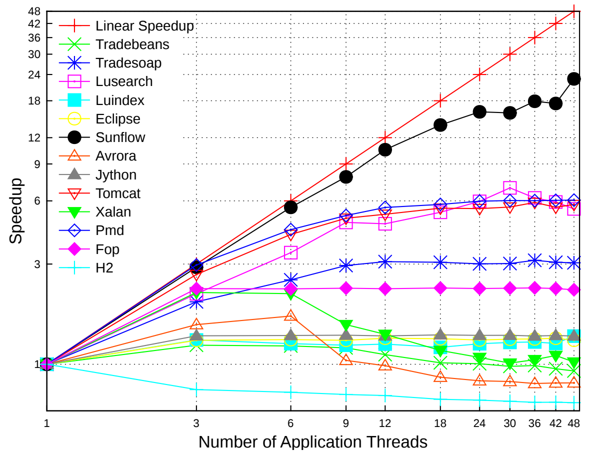


Figure 1: Scalability comparison of DaCapo benchmarks with large heap size.

2. EXPERIMENTS

Parallel garbage collectors came into existence when multiprocessors were simpler in design. However contemporary hardware is much more complex. Therefore, it is time to re-evaluate their scalability. We conducted experiments to evaluate (1) GCs effect on application scalability, and (2) GC scalability. The experimental setup and the results of the experiments are described next.

2.1 Setup

Hardware and Operating System.

With the introduction of more and more cores, the shared Front Side Bus (FSB) doesn't scale due to high contention. To overcome this, cores are grouped into nodes which are connected by a sophisticated network topology such as mesh or ring. In these architectures, a typical snooping based cache coherency algorithm becomes highly inefficient as it needs broadcast of cache messages. Therefore, to solve this, the physical memory is statically partitioned among the nodes, with each node being in-charge of one physical memory partition. When a core accesses a remote address, i.e. for which its node is not in-charge, it forwards the request to the node in charge of the corresponding memory partition. Such remote access is substantially more expensive than accessing the local memory. These architectures are commonly known as Non Uniform Memory Access (NUMA).

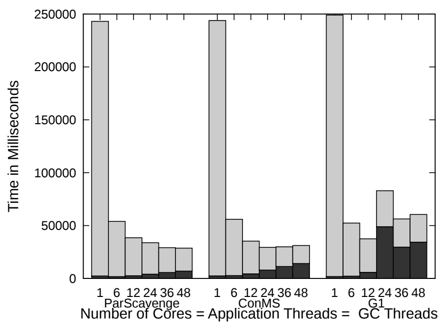
To experiment on such a NUMA many-core machine, we conduct all our experiments on a machine with four *AMD Opteron 6172* sockets, each consisting of 2 nodes. Each node consists of 6 cores (2.1GHz clock rate), a 6MB shared L3 cache (48-way set associative) and one memory controller (1333MHz memory speed). The nodes are connected by HyperTransport links (up to 25.6 GB/s at 6.4 GT/s per link) with a maximum hop count of 2. In total there are 48 cores, 32GB of RAM consisting of 8 memory nodes and 48MB of L3 cache. Each core consists of a 512KB L2 cache (16-way set associative) and L1 instruction and data caches, each 64KB (2-way set associative). All caches use 64-byte cache lines. The system runs a Linux 2.6.35 64-bit kernel from Ubuntu 10.10.

Managed Runtime Environment.

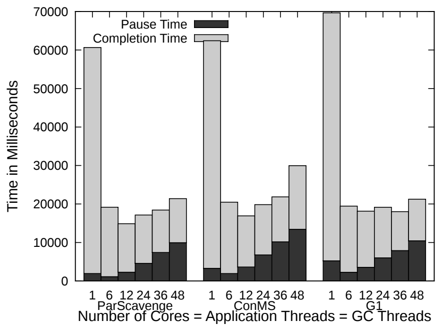
For the experiments we use OpenJDK version 7 configured in server mode [14]. It provides three parallel/concurrent GCs: (1) The default GC of OpenJDK, a generational parallel scavenge GC (ParScavenge), consisting of a parallel copying young generation collector and a parallel compacting GC for the old generation; (2) The generational concurrent mark-sweep GC (ConcMS), consisting of a parallel copying young generation and a concurrent mark-sweep old generation [8]; and (3) Garbage First (G1) [5], which is a concurrent, incremental, parallel garbage collector with an assumption of the region used for allocation being the young generation (please see section 4 for a detailed description).

Benchmarks.

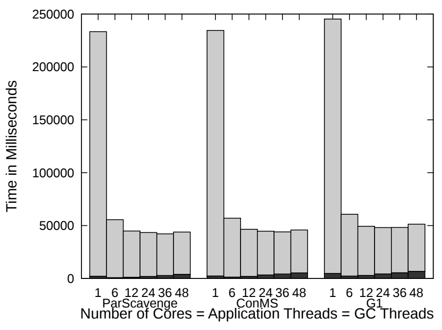
In order to evaluate the scalability of GC, it is important to be sure that the scalability issues come from the GC and not the application. Therefore, it is essential to ensure that the benchmarks chosen for the experiments are themselves scalable. To ensure this, we tested all the benchmarks of Dacapo suite. Figure 1 shows the comparison of speedup, i.e., the execution time with one thread compared to the execution time with n threads, of all Dacapo benchmarks. This experiment was performed with a very huge initial heap size (26GB) to avoid triggering GC as much as possible. Furthermore, in order to emulate a hardware with same number of cores as application threads in the experiment, we set CPU-affinity for each run such that all the threads are packed in minimum number of required nodes. The experiments were



(a) Sunflow



(b) Lusearch



(c) Tomcat

Figure 2: **Application Scalability:** Impact of garbage collection on the application performance when the number of core increases.

repeated three times and the average execution time is reported. The figure shows that *Sunflow*, *Lusearch*, *Tomcat* and *Pmd* are the most scalable benchmarks. Due to lack of space, we only report the results of first three benchmarks.

For all our other experiments, we used 100MB as the maximum heap size to emulate the long-running behavior. The tests were repeated 10 times, and mean values are plotted.

2.2 Results

We used the stop-the-world pause time as our metric of GC performance. We modified the safepoint handling code, which executes a given GC operation with application threads stopped[3] at safepoints, to record the stop-the-world pauses done by the GC. We also added counters in the VM initialization and exit code to fetch the total execution time.

GC acts as a bottleneck for the application. Figure 2 illustrates the impact of GC performance on application scalability. The figure plots total execution time and stop-the-world pause time as a function of the number of cores. To emulate a hardware with the specified number of cores, the number of application threads and GC threads were set to be equal to the number of cores and were pinned to the cores such that the minimum required nodes are used.

As can be seen, execution time doesn't decrease, except in few cases. Furthermore, the proportion of pause time in the execution time keeps increasing after a certain number of cores. At 48 cores, on Sunflow, it can be as high as 24% with ParScavange, 45% with ConcMS and 58% with G1. a similar trend is visible on the other two benchmarks as well. This also shows that, if the pause time had been decreasing (or at least non-increasing), the application would have scaled better. Therefore, we conclude that the GC is critical to application scalability and acts as a bottleneck.

OpenJDK GCs do not scale. Figure 3 plots the stop-the-world pause time and total execution time as a function of the number of GC threads. The experiment evaluates GC scalability. Therefore, we keep the number of application threads and cores constant at the maximum, i.e., 48. Pause times decrease until a certain number of GC threads is reached, then monotonically increase. Although the amount of work during garbage collection cycles is the same and the number of GC threads is increasing, pause time still keeps increasing. Therefore, we conclude that garbage collectors of OpenJDK don't scale.

[3]A point during program execution at which all GC roots are known and all heap object contents are consistent.

3. FACTORS AFFECTING SCALABILITY

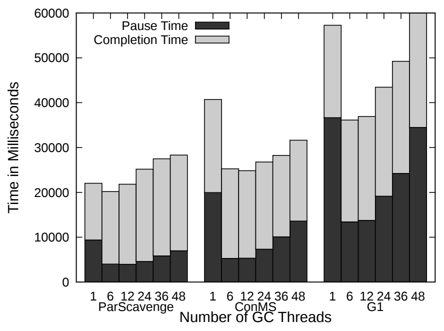
Given the above results, we seek to understand the possible factors affecting the scalability of garbage collectors. For this, we instrumented the OpenJDK code to collect information related to object locality. We also experimented with the load balancing scheme used by the GCs in OpenJDK. Due to lack of space, we present experiment results of ParScavenge GC only, on the three benchmarks with number of cores, GC threads and application thread set to 48. The results with other collectors are similar. The following factors were observed to be causing problems:

Remote Scanning of Objects. Live objects are accessed many times during garbage collection, such as while marking, copying and updating references after copying. Due to the lack of affinity between the GC thread accessing an object and the memory node where it is allocated, the chance of a remote object scan are 7 out of 8 with 8 memory nodes. Since remote access is much more expensive than a local one, it can severely affect the performance. Furthermore, contented access by multiple cores on the same inter-connect link makes them wait for transfer to complete. This waiting time is proportional to the number of cores and hence affects the scalability. Table 1 shows the proportion of object scans which are remote. During marking and copying, approximately 87% of object scans are remote, compatible with 7 out of 8 hypothesis. Therefore, by increasing memory locality during the scanning phase, we expect an improvement.

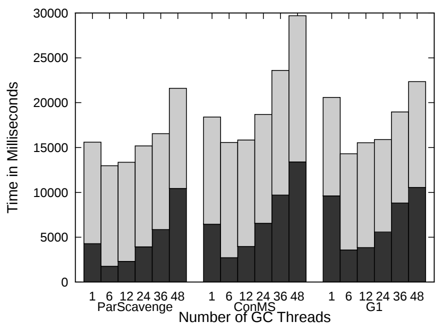
Remote Copying of Objects. The current implementation of garbage collectors in OpenJDK copies objects into a GCLAB (GC Local Allocation Buffer) which is local to one GC thread to avoid locking for every in-copy object. The idea is similar to the Thread Local Allocation Buffer (TLAB), a fraction of young generation space alloted to every application thread for lock-free object allocation in the

Locality Factor(%)	Sunflow	Tomcat	Lusearch
Remote Scanning	87.86	87.03	87.79
Remote Copy	83.35	78.44	82.73

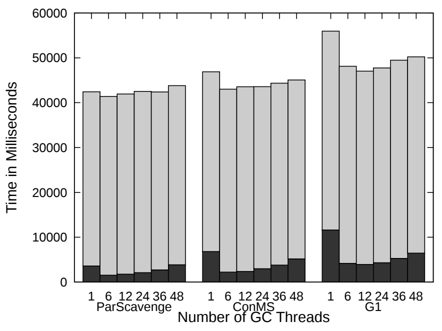
Table 1: Proportion of remote access during collection.



(a) Sunflow



(b) Lusearch



(c) Tomcat

Figure 3: **GC Scalability:** Performance of the garbage collectors when the number of GC threads increases.

fast-path. However, there is no guarantee that the physical memory for the GCLAB in use by a GC thread comes from the local memory node. In addition, like remote scanning, lack of object affinity between GC thread and in-copy object also causes this. Table 1 shows the proportion of objects which are copied remotely during garbage collection. Approximately 83% of object copies are remote on Sunflow. The numbers are similar for the other two benchmarks as well. Therefore, we expect better performance by gaining more object locality during copying.

Load Balancing does not Scale. OpenJDK implements a load balancing technique called *work stealing* [7]. In this, each GC thread fills its own work-list during root scanning. After processing all the elements in its work-list, the thread starts stealing objects from other GC thread's work-lists. We performed a test to count the number of objects in the work-lists of all the GC threads after root scanning. We observe that the load is highly imbalanced. Some work-lists contained no objects at all, whereas others were full.

To find that work stealing was a problem, we profiled the OpenJDK code with OProfile [17], and found that function, *steal_best_of_2*, which performs the stealing, is one of the most time consuming functions. The function selects, at random, 2 work-lists used by 2 GC threads, compares the size of the work-lists and, takes an work-item from the bigger one. The *size* parameter acts as the cause of contention as, it is also being frequently accessed by the owner of the work-list (for adding and removing work-items) and probably by other GC threads (to steal the work-items) as well, causing a ping-pong effect for the cache line containing the parameter. Furthermore, the number of invocations of this function increases with the number of GC threads.

We performed a test with work stealing disabled, and measured the difference. Table 2 compares pause times, with and without work-stealing. We observe an improvement of up to 33% in pause time with 48 GC threads, even though the work-lists remained highly imbalanced. A different, more scalable load balancing algorithm would likely to give even better results.

Benchmark	Pause Time (ms) With stealing	Pause Time (ms) Without stealing	Reduction(%)
Sunflow	6993	6884	1.55
Tomcat	3857	2892	25.01
Lusearch	10435	6960	33.30

Table 2: Effect of disabling work stealing on the pause time.

4. RELATED WORK

In order to exploit the available parallelism in the hardware, many parallel garbage collection techniques have been proposed [3, 5, 10]. However, these designs don't show any evidence of being topology aware. Therefore, it is unlikely that they will scale well on NUMA hardware.

There is a considerable amount of work on efficient load balancing among worker threads (a group of threads working on a shared set of tasks to complete). Flood et al. [7] propose a per-object work stealing technique using lock-free double ended queues (also used in [11]). This is the scheme implemented in OpenJDK, and Section 3 shows that it doesn't scale well. Oancea et al. [12] associate work-lists to dedicated partitions of the heap. Only the owner of a work-list (the worker thread that works exclusively on that work-list) traces the objects in the corresponding heap partition. For load balancing, workers take ownership of whole work-lists, rather than of individual objects. However, we believe, the work-list ownership scheme doesn't honor the hardware topology. Therefore, causes higher access penalties when a worker takes the ownership of some heap partition located on a remote memory node.

There has been much research on concurrent garbage collection, mainly targeting real-time applications. Detlefs et al. [5] propose *Garbage First*, a solution achieving real-time goals with high probability. It partitions the heap into fixed-size regions, each with a remembered set recording pointers from the rest of the regions. With this, any set of regions can be chosen for collection, in order to ensure shorter evacuation pauses. It is implemented in OpenJDK and has been evaluated in this work. Pizlo et al. [15] propose *STOPLESS* which provides a combination of concurrent mark-sweep algorithm for collection, and a concurrent compaction scheme for defragmenting the memory. In another work [16], Pizlo et al. propose *Schism*, a fixed-size allocation technique to avoid external fragmentation, completely avoiding the problem of compaction. Marlow et al. [11] propose to split the heap into per processor local heaps, and a global heap, to avoid synchronization between the threads during local collections. However, to let other threads proceed during a local collection, the collection thread avoids copying objects, and instead use a flag to indicate if the object belongs to a local heap or the global heap. None of the algorithms explained in this paragraph are NUMA-aware and hence are unlikely to scale on such hardware.

Parallel Scavenge and Concurrent Mark Sweep are the other two GCs evaluated in this work. Parallel Scavenge [8] uses a parallel copying GC for young generation which consists of an *Eden* space and 2 survivor spaces called, *To* and *From*. At every GC cycle, all the live objects in *Eden* and *From* space are copied to *To* space until its full and the rest are promoted (copied) to the old generation. Finally, *To* and *From* spaces are flipped. For the old generation, it uses a parallel *mark-sweep-compact* collection algorithm. Concurrent Mark Sweep [8] uses a GC very similar to that of Parallel Scavenge for young generation. For the old generation, it uses a concurrent mark-sweep algorithm with only initial mark and remark phase being stop-the-world. To deal with fragmentation, it splits or joins the available free blocks to meet the demand.

Except [11], all the work mentioned so far in this section has been evaluated on at most a eight-core machine and a maximum two memory nodes. This is insufficient to expose the problems which arise in contemporary many-core systems. We are unaware of any thorough evaluation of GCs on large scale multi-core machines.

A NUMA-aware young generation GC exists in OpenJDK as an extension to ParScavenge GC[4]. It ensures that a TLAB, allocated to a thread comes preferably from its local memory node. However, this doesn't solve the problem of remote object scanning and copying. Ogasawara et al. [13] and Tikir et al. [18] ensure that an object is copied to the memory node where its accessed most of the time. Although this improves access latency for mutators, it worsens the garbage collection.

Majo et al. [9] prove that optimizing only for data locality can conflict with the benefits of cache contention avoidance, and vice-versa. They suggest that system software must take into account both data locality and cache contention, and that memory management and process scheduling should be aware of each other.

5. CONCLUSION AND FUTURE WORK

In this paper, we presented some preliminary results, obtained by experimenting GCs implemented in OpenJDK7 on a 48-core NUMA machine. They suggest that GC acts as a bottleneck for applications and does not scale. Rather than pause time decreasing, we observed an increase as the number of cores increases. We explored some of the factors affecting the scalability of the GC.

In future work, we plan to fix the above mentioned factors, and see if this solves the scalability issue. A similar approach has been proven effective for improving scalability of Linux kernel in [4]. In case this does not work, then a new design, probably one which resembles a distributed GC [6], will be required. We would like to explore the idea of accessing shared objects using message passing rather than shared memory. This will enable us to perform processor local heap collection in complete isolation from the other application threads. The similar idea is used in Barrelfish [1] to improve OS scalability.

[4]We have not found any documentation on this, nor we could evaluate the code because of a Linux bug (Redhat bug 727700).

6. REFERENCES

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *SOSP*, 2009.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, 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. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, 2006.

[3] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *PLDI*, 2008.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *OSDI*, 2010.

[5] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *ISMM*, 2004.

[6] P. Ferreira and M. Shapiro. Larchant: Persistence by reachability in distributed shared memory through garbage collection. In *ICDCS*, 1996.

[7] C. H. Flood, D. Detlefs, N. Shavit, and X. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Symposium on JavaTM Virtual Machine Research and Technology Symposium*, 2001.

[8] Whitepaper: Memory management in the Java Hotspot virtual machine. `http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf`, 2006.

[9] Z. Majo and T. R. Gross. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. In *ISMM*, 2011.

[10] S. Marlow, T. Harris, R. P. James, and S. P. Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM*, 2008.

[11] S. Marlow and S. P. Jones. Multicore garbage collection with local heaps. In *ISMM*, 2011.

[12] C. E. Oancea, A. Mycroft, and S. M. Watt. A new approach to parallelising tracing algorithms. In *ISMM*, 2009.

[13] T. Ogasawara. NUMA-aware memory manager with dominant-thread-based copying gc. In *OOPSLA*, 2009.

[14] Openjdk7. `http://jdk7.java.net/`.

[15] F. Pizlo, D. Frampton, E. Petrank, and B. Steensgaard. STOPLESS: A real-time garbage collector for multiprocessors. In *ISMM*, 2007.

[16] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: Fragmentation-tolerant real-time garbage collection. In *PLDI*, 2010.

[17] Oprofile. `http://oprofile.sourceforge.net`.

[18] M. M. Tikir and J. K. Hollingsworth. NUMA-aware Java heaps for server applications. In *IPDPS*, 2005.