COLD OBJECT IDENTIFICATION AND SEGREGATION VIA APPLICATION PROFILING

by

Abhijit S. Taware

Bachelor of Engineering in Information Technology, Pune University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Kenneth B. Kent, PhD Computer Science
	Gerhard W. Dueck, PhD Computer Science
Examining Board:	Suprio Ray, PhD, Computer Science, Chair
	Michael W. Fleming, Ph.D, Computer Science
	S. A. Saleh, Ph.D, Electrical Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

July, 2019

Abstract

Managed runtimes like the Java Virtual Machine (JVM) provide automatic memory management, i.e., garbage collection, to remove the burden of explicitly freeing allocated memory from the programmer. Garbage collectors (GC) periodically walk the heap and free unused memory. Some of the surviving objects are infrequently accessed. However, the JVM has to account for these objects during each GC cycle, which is clearly an unnecessary overhead. Such objects can be categorized as cold and moved to a dedicated memory area. This segregation gives an opportunity to perform the GC either on hot or cold regions. Typically only hot regions are GCed. This leads to having fewer objects to process during each GC cycle and hence exhibit better performance. Furthermore, cold objects can be stored in cheaper and larger capacity memory, like NVRam, leaving more main memory for the hot objects of the application. Identification of such objects is a particularly difficult task. As a part of this thesis, application profiling is explored to identify cold classes. Various object properties are evaluated and alternate methods are studied to overcome the limitations with application profiling.

The experimental results show 4% average runtime gain with applications showing predictable behavior and object access patterns. Memory intensive applications like in-memory database showed better object segregation. A very primitive technique of operating system supported memory protection was used to further assist cold object identification. A better alternative to memory protection will offer significant gains.

Dedication

To my parents, my wife Komal, daughter Anaya and son Neil.

Acknowledgements

This research was conducted within the Centre for Advanced Studies—Atlantic, Faculty of Computer Science, University of New Brunswick. The author is grateful for the colleagues and facilities of CAS Atlantic in supporting the research. The author would like to acknowledge the funding support of the Natural Sciences and Engineering Research Council of Canada (NSERC), 503509-16. Furthermore, I would also like to thank the New Brunswick Innovation Foundation for contributing to this project. This document, and the work which came from it, would not be possible without the guidance and support of family, friends and colleagues. A special thanks to my supervisors Dr. Ken Kent and Dr. Gerhard Dueck for their continued feedback and insights, and my colleagues in the CASA IBM project who helped with introducing me to the research field and advice on my continued academic career.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern programming languages support automatic memory management, consisting of garbage collection (GC), which offers the following benefits.

- Allows a programmer not to worry about memory management
- Increases readability of source code
- Decreases development time and costs
- Avoids memory leaks

These benefits come at the cost of performance. The language runtime consists of an extra layer of abstraction between the operating system and the application. The runtime is composed of various component like an interpreter, garbage collector, Just-In-Time (JIT) [2] compiler, etc, which contribute to the overhead. A better understanding of such complex subsystems also reveals opportunities to improve performance. For example, the Java Virtual

Machine(JVM) [3] allocates and frees objects on the heap. The objects allocated in memory have varying lifetimes. Many objects are short-lived. This fact has been exploited by generational garbage collectors [4]. The relatively fewer objects that live for a long time are moved to the tenured area, which is garbage-collected less frequently. Among the objects that persist for a long time, there are some accessed infrequently, i.e., cold objects[5].
Generally, cold objects are co-located with other active objects in the heap, which pose following issues.

- Active objects prevent memory pages from being swapped out, leaving cold objects in both memory and cache.
- Garbage collector traverses all the cold objects along with hot ones, which is an unnecessary overhead.

Identifying and segregating cold objects to a different memory region would help improve application performance due to the following reasons.

- Cold regions can be garbage-collected less frequently.
- Improve the real memory footprint and the cache coherence [5].

Segregation also gives another compelling opportunity to use a different memory device for storing cold objects, i.e., a cheaper and slower option like persistent memory [6] is an attractive choice. Persistent storage is also termed NVRAM (Non-Volatile Random Access Memory), which is less costly than

dynamic RAM with slightly higher access time, e.g., Intel 3D Xpoint memory technology [7][8][9][10]. The main application for NVRAM has been in the area of database transactions [11] where frequent small data accesses are common.

An ideal solution would correctly identify and segregate cold objects. Identifying cold objects at runtime directly affects application performance, so an approximation is required [5]. The use of secondary memory as cold storage incurs more read and write overhead [10], further affecting the application performance. The net performance gain will be achieved if the solution offsets these overheads. An offline approach, i.e., application profiling, is explored in this thesis as an alternative to identify and segregate cold objects. The proposed solution is built using a page-protection technique [12], which is modified to protect only the older regions. The experimental evaluation showed gains with applications having predictable behaviour and high percentage of cold objects. The static profiling performs poorly and needs to be augmented with careful runtime analysis.

The rest of the thesis is structured as follows.

- *Chapter 2*: Provides the information about Java Virtual Machine (JVM) architecture [3], basic garbage collection algorithms and the benchmarks used to evaluate the proposed solution.

- *Chapter 3*: Gives a detailed definition of cold objects, OpenJ9 GC policies are discussed and the suitability of the balanced GC policy for

this work is elaborated.

- *Chapter 4*: Various techniques for measuring the object temperature are discussed.

- *Chapter 5*: Delves deeper into the application profiling technique, which is the chosen object segregation technique for the proposed research.

- *Chapter 6*: Provides information about the experimental setup and explains the results for various benchmarks.

- *Chapter 7*: Concludes with the results and provides directions for future work.

Chapter 2

Background

Traditional programming languages, like C and C++ compile the source code into binary machine code. These languages require programmers to explicitly allocate and free memory. However, many programming languages are interpreted and support automatic memory management. Interpreted languages are compiled into machine independent bytecode. A virtual machine (VM) executes the program by interpreting the bytecode. Automatic memory management frees the allocated objects that are no longer in use, i.e., garbage collection. This chapter will introduce the fundamental concepts of Java [3], the Java Virtual Machine [13], garbage collection (GC), and benchmarks [14] [15].

2.1 Java

Java [3] is an interpreted programming language with dynamic memory management. Java programs are compiled into Java bytecode. The bytecode is interpreted by a Java virtual machine, which allows Java programs to run on any platform. Java being object-oriented, memory allocations happen on an object basis.

2.2 OpenJ9 Java Virtual Machine

The Java Virtual Machine (JVM) [13] manages system memory and provides a portable execution environment for Java applications. OpenJ9 [16] is an open source JVM by IBM, which is fully compliant with the Java Virtual Machine specification [13]. OpenJ9 boosts application performance by lowering memory footprint, providing a quick startup and high application throughput. IBM OpenJ9 is used for the work in this thesis and Figure 2.1 depicts the JVM components.

2.2.1 Class Loader

The class loader is responsible for loading, linking and initializing the class file when it is referenced for the first time. The following three tasks are perfomed by the class loader.

1. Loading — This component loads the classes, which are further classi-

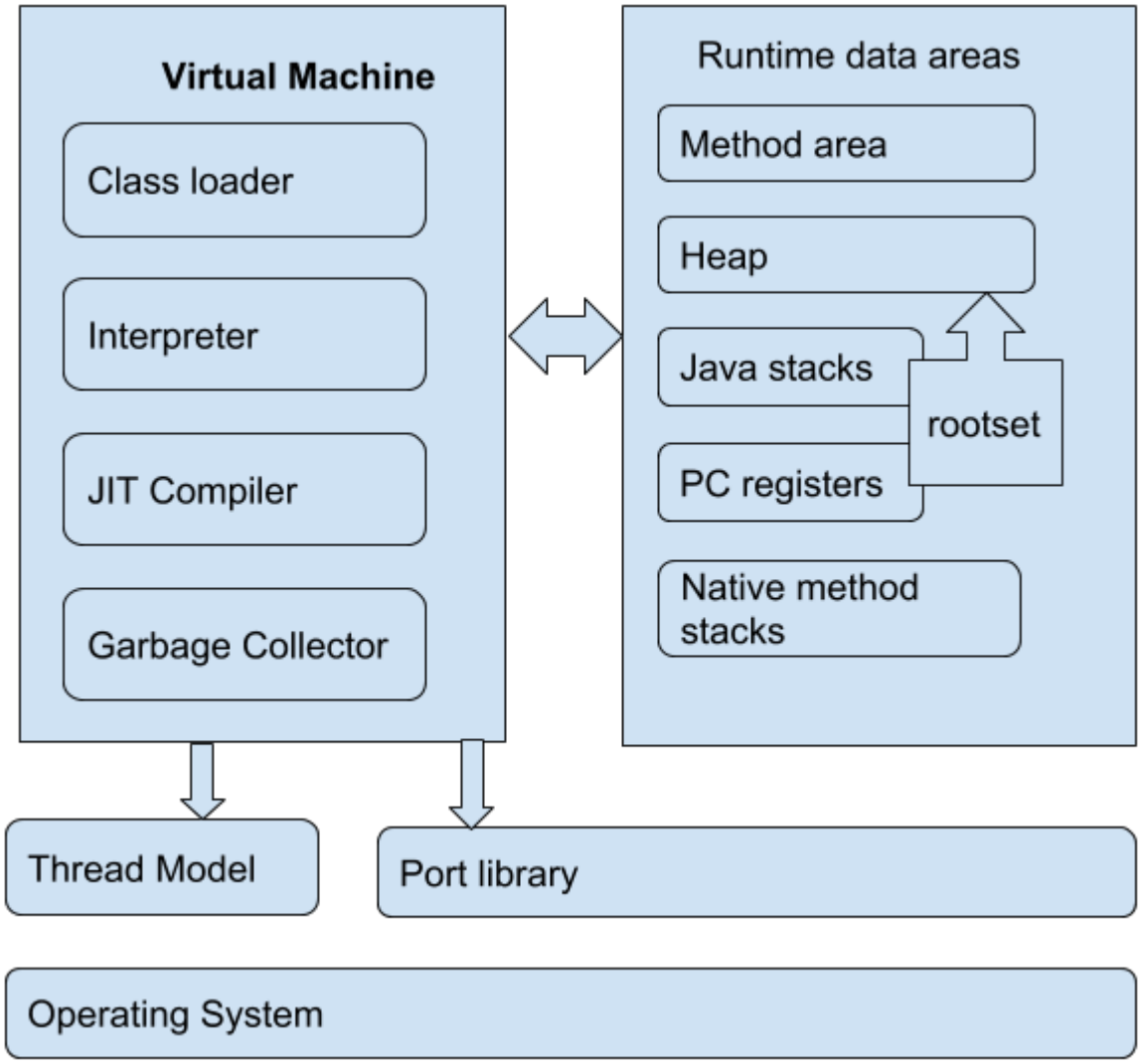

Figure 2.1: OpenJ9 Architecture

fied as bootstrap, extension, and application class loaders.

2. Linking — This phase verifies the bytecode, allocates static variables, and resolves references.

3. Initialization — All static variables are assigned their original values and corresponding static blocks are executed.

2.2.2 Allocator

The allocator is a critical component of the JVM and pertinent to understanding garbage collection. It manages pools of free memory and allocates objects in the Java heap at the request of applications, class libraries, or the VM. Each allocation requires a heap lock [17] due to the shared nature of the heap. The heap lock is released after allocation. The lock is also released when allocation fails due to insufficient memory, in which case a garbage collector (GC) is triggered. Heap locks are expensive, hence small objects are allocated to thread local heaps (TLH) [18]. TLH are per-thread reserved caches in the heap that help allocate objects without any locks and hence are faster.

2.2.3 Garbage Collector

The garbage collector (GC) [13] frees the memory occupied by dead objects. Any object that is unreachable from the root set is considered dead and hence garbage collected. The root set is a set of references to the contents of the

stacks and registers of the JVM threads and other internal data structures. Various GC policies will be discussed after explaining runtime data areas.

2.2.4 Just-In-Time (JIT) Compiler

The Just-In-Time (JIT) [2] compiler improves the performance of Java applications by compiling platform-neutral Java bytecode into native machine code at run time. OpenJ9 records the number of times a method is called. After the method invocations count reaches a pre-defined threshold, JIT compilation is triggered. The JIT-compiled methods do not need to be interpreted and hence are faster. The methods can be compiled at different optimization levels: cold, warm, hot, very hot, or scorching.

2.2.5 Port Library

The port library [19] is a thin native layer that isolates the use of OS-specific services and resources for handling memory access, file management, threading, IP sockets, locks, I/O, and interrupts. This protects the higher layers of the class libraries from having to know the details of the underlying platform.

2.2.6 Runtime Data Areas

The runtime data areas are allocated using the port library and it consists of the following components.

1. Method area — It hosts class metadata (constants and static variables)

and bytecodes for methods and constructors. The runtime populates this area whenever the application uses a new class. The runtime constant pool is also a part of the method area.

2. Heap — This is a memory space shared among all Java threads. All the dynamic objects are allocated using heap memory.

3. Java stacks — Each JVM thread owns a private JVM stack that is created when the thread is launched. When a thread invokes a method, the method's local variables are stored in a frame on the invoking thread's Java stack. The JVM manages the stack only by pushing or popping the frames.

4. Root set — The Java stacks and PC registers hold references to objects allocated on the heap. These references collectively form the root set. Any object that is unreachable from the root set is treated as dead and collected during the GC cycle.

5. Native method stack - A non-Java method is served using the native method stack and each thread owns a separate copy.

2.3 The Java Class

The class consists of member variables and methods. The variables can be primitive data types or objects, i.e., references. The reference variables are also known as object slots. Inside the OpenJ9 [16] implementation, a

Java class, when loaded, is transformed into ROM and RAM Classes. The transformation is done based on the mutability of the information in the class file. The immutable information is stored under ROM Classes and the rest of the information goes to a RAM Class as explained below.

- ROMClass — It contains all of the class's immutable data, e.g., field shape descriptions, method bytecodes, annotation data and UTF8 strings (e.g., class name).
- RAMClass — It contains mutable data, such as static class variables.

The immutability allows ROMClass to be shared between all the instances of a class, whereas each instance of a class has a separate RAMClass.

2.4 OMR

OMR [20] is a common set of core runtime technology components, e.g., garbage collection, port library, and a JIT compiler that can be used to build robust language runtimes. The garbage collection library allows the runtime developers to reuse GC algorithms, whereas the port library provides an OS independent abstraction for common system calls.

2.5 Basic GC Algorithms

The GC algorithms described below are managing heap memory and are categorised as tracing or non-tracing collectors. All tracing collectors consist

of the following steps.

- Find all objects that are reachable (live) from the root set. This is known as the 'mark phase'.
- Reclaim memory used by all the unreachable (dead) objects.

Reference counting is a non-tracing collection algorithm, which uses counters to avoid tracing and is described in the following section. GC algorithms can be evaluated based on the following key parameters [21]:

- Throughput — total time available to the mutator threads, i.e., the application threads
- GC overhead — the measurement of processor time used by the garbage collector
- Pause time — amount of time an application is stopped during GC
- Frequency of garbage collection
- Footprint — maximum heap size
- Promptness — the time between when an object becomes dead and when the memory becomes available
- Completeness — all the garbage has to be reclaimed eventually

2.5.1 Reference Counting

Reference counting [22] is one of the earliest methods of garbage collection. Every object maintains a reference count. It represents the number of incoming references to the object. The object is reclaimed when the counter reaches zero. Though counter updates are required on every pointer update, the memory is reclaimed immediately. Objects from different regions that refer to each other directly or indirectly through a chain of references form a cycle. Reference counting fails to GC such objects, which can be reclaimed using the mark-sweep technique explained next.

2.5.2 Mark Sweep

The objects reachable from the root set are marked [23] as alive. This is termed the mark phase. During the sweep phase, remaining objects are garbage collected. Fragmentation is an inherent issue with this scheme. Memory fragmentation occurs when most of the memory is allocated in a large number of non-contiguous blocks, or chunks - leaving a good percentage of total memory unallocated.

2.5.3 Mark Compact

Live objects are identified as a part of the mark phase [24]. All live objects are moved to the beginning of the heap, squeezing out garbage, and maintaining the original allocation order in the heap. Though the fragmentation issue

is addressed, object movement requires many pointer updates, which makes implementation computationally intensive.

2.5.4 Copy Collector

The heap is split into two semi-spaces [25]. Allocation happens until the semi-space is full, after which GC starts. The GC copies all reachable objects to the second (empty) semi-space. All objects in the first semi-space are discarded, i.e., the first semi-space is treated as empty. Finally, the roles are switched for these semi-spaces. Though faster allocation is achieved, half the heap space is wasted.

2.5.5 Generational GC

The generational hypothesis, i.e., most objects die young, is exploited in this approach [26]. The heap is split into a young space and a few older spaces. Allocations happen to the young generation, whose space is collected more frequently. Surviving objects are promoted to older generations. In addition to stack references, the root set also includes references from other generations also known as the remembered set. The GC cycles are faster since less memory is examined. Many of the examined objects are collected. This leads to shorter pause times and better application response.

2.6 Benchmarks

Java benchmarks measure the performance of various JVM subsystems like the JIT compiler, bytecode interpreter, garbage collector, etc. The following sections explore the DaCapo[14] and cloudGC [15] benchmarks.

2.6.1 DaCapo

The DaCapo benchmark suite [14] [27] is highly demanding in terms of memory requirements and hence useful to test garbage collectors. A brief description of the DaCapo suite [27] used for benchmarking OpenJ9 is given below.

- avrora — simulates a number of programs run on a grid of AVR microcontrollers. It contains a flexible framework for simulating and analyzing assembly programs, providing a clean Java API and infrastructure for experimentation, profiling, and analysis.

- fop — takes an XSL-FO (XSL Formatting Objects) file, parses it and formats it, generating a PDF file. XSL-FO is a markup language for XML document formatting that is most often used to generate PDF files.

- h2 — executes a JDBCbench-like [28], in-memory benchmark, executing a number of transactions against a model of a banking application.

- jython — interprets the pybench Python benchmark.

- luindex — uses lucene [29], a high-performance text search engine library, to index a set of documents comprised of the works of Shakespeare and the King James Bible.
- lusearch — uses lucene to do a text search of keywords over a corpus of data comprised of the works of Shakespeare and the King James Bible.
- pmd — analyzes a set of Java classes for a range of source code problems.
- sunflow — renders a set of images using ray tracing.
- xalan — transforms XML documents into HTML.

Most of these benchmarks are multi-threaded and reflect common usage of deployed Java applications.

2.6.2 CloudGC

CloudGC [15] is a GUI-based Java EE benchmark that focuses on stressing the underlying runtime and GC. It is suitable for deployment on Platform as a Service clouds. CloudGC maintains an in-memory object graph. Each object contains an object payload and a set of references to other objects. The root set is simulated by maintaining stack-frame-like structures. The servlet endpoints allow the user to access/mutate references. Each request has its own stack and a shared stack is maintained as well. This model takes

care of the simulated generational hypothesis that most objects die young. Some of the available actions with cloudGC are discussed below:

- add/remove top frame
- allocate object
- change reference
- read/write object payload

The cloud service providers need to fulfill certain Service Level Objectives (SLOs), e.g., network latency, hardware resources, number of transactions, etc. Any SLO violations could entail a financial penalty. The GC being a computationally expensive task can affect SLOs. Existing Java EE cloud benchmarks like CloudTrader or TPC-W [30] are not designed to specifically stress the GC component of the runtime. The CloudGC benchmark experimentally evaluates the performance of the four GC policies (Gencon, Balanced, Optavgpause and Optthroughput) available in the IBM J9 Java Runtime, which makes it an essential tool in evaluating the solution proposed in this thesis.

2.7 Summary

This chapter introduced required background which is further referred as follow.

- Mark-compact and copy collectors provide an opportunity for object segregation, which is further elaborated in section 5.4.1
- Section 6.3 depicts the results of the DaCapo benchmark discussed in this chapter.
- The CloudGC benchmark is instrumental in understanding the nature of long-running applications, which was used for the evaluation of the page protection solution mentioned in Section 4.3.

Chapter 3

Cold Objects and Balanced Policy

This chapter defines cold objects and explores their significance for the proposed research. Various garbage collection policies are analyzed, considering object temperature, and the balanced garbage collection policy is discussed in detail.

3.1 Cold Objects

An object is an instance of a class, allocated on the heap. Memory occupied by dead objects is reclaimed during garbage collection cycles. Depending on the policy, the GC is triggered for the entire heap or a region of it. Some of the objects are accessed very frequently while others show very low activity.

The access frequency decides the temperature of an object. The infrequently accessed objects are termed cold. Figure 3.1 shows a hypothetical representation of the segmented heap with the following memory cell types.

- Empty — Free memory
- Dead — Objects unreachable from the root set, should be garbage collected
- Hot — Frequently accessed live objects
- Cold — Infrequently accessed live objects

As shown in the first row of Figure 3.1, the cold and hot objects co-exist in the heap, which affects cache performance. The presence of cold objects in a cache or a page frame reduces the number of objects in cache or memory, respectively, that are likely to be referenced again, i.e., reduced cache or page hit rates. Thus, cold objects surviving multiple GC cycles are clearly generating overhead. Segregation of cold and hot objects to different memory regions is shown in the second row of Figure 3.1, which directly helps with cache performance. The garbage collector has to account for fewer objects if the GC is done only for the hot or cold region, as depicted in the last two rows. The segregation also allows the garbage collector to make a more informed decision about selecting regions for GC.

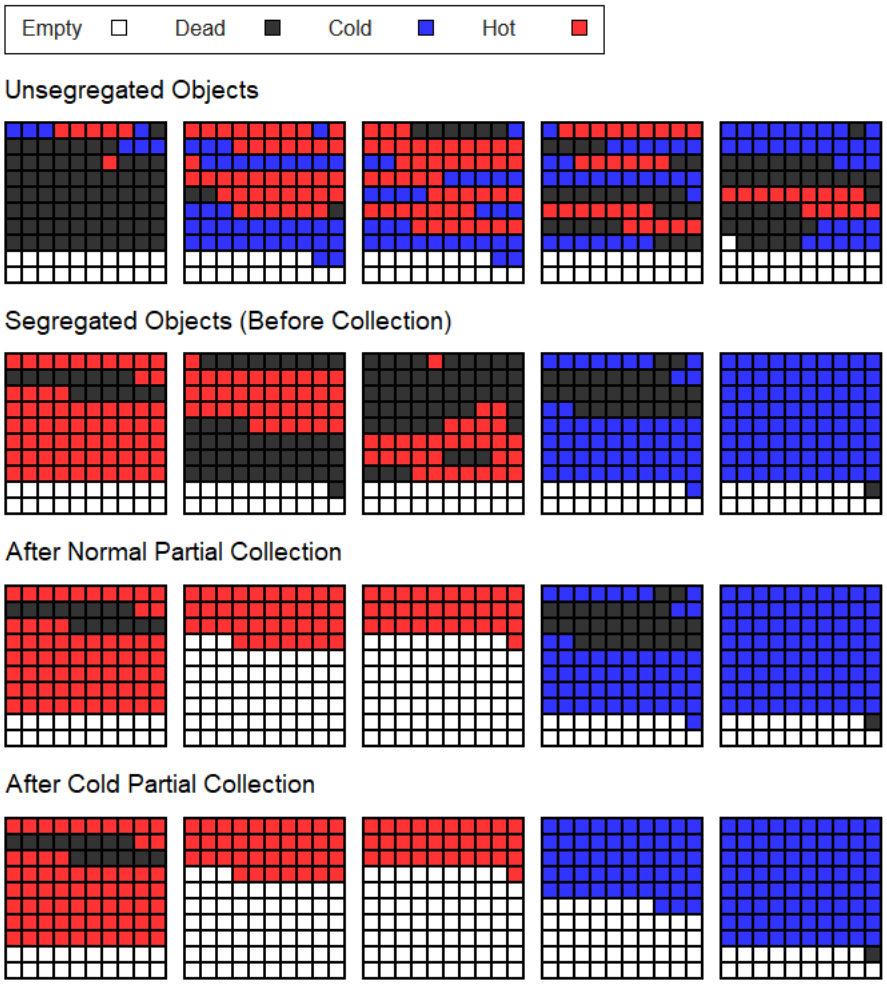

Figure 3.1: Temperature-based segregation and GC

3.2 Non-Volatile Random-Access Memory

Object usage patterns differ based on object temperature. Hence, cold objects, being infrequently accessed, can be moved to a cost-effective memory alternative. Non-Volatile Random-Access Memory (NVRAM) provides such an alternative, e.g., the Intel 3D X-Point memory chip. NVRAM provides data persistence across power cycles. It is more performant than a hard disk and solid-state disks, but it is slower than random access memory (RAM). The cost is significantly lower than the RAM. Figure 3.2 depicts persistent memory, i.e., NVRAM sitting between DRAM and hard-disk. The speed and cost go up along the y-axis, while the capacity goes down.

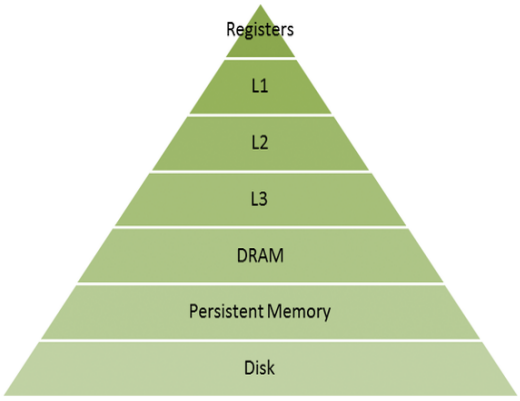

Figure 3.2: Memory hierarchy

3.3 OpenJ9 GC Policies

The OpenJ9 Java Virtual Machine supports many policies that control the behavior of the garbage collector. Knowledge of the concurrent marking phase is crucial for a better understanding of IBM's OpenJ9 GC policies. The concurrent mark phase helps reduce the pause times. Each thread scans its own stack to build a root set. The root set is used to trace the live objects. A low-priority background thread does the tracing work during heap-lock allocation [17]. Heap-lock allocation requires a lock and is therefore avoided, if possible, by using the cache. Due to the concurrent nature of marking, any updates done to already marked objects must be tracked. During the stop-the-world phase, the GC uses card tables [31] to retrace updated references. Card tables use write barriers [32] requiring an additional test on every pointer update, which increases processor overhead. Concurrent marking gives reduced pause times, at the expense of mutators doing extra tracing. The OpenJ9 policies discussed below optionally use concurrent marking.

1. OptThruput — Concurrent marking is disabled. The application stops during GC. Useful for large-heap applications, that favor high throughput more than short GC pauses.

2. OptAvgPause — Uses concurrent marking to reduce the application pause time. Useful for applications that are sensitive to response time caused by GCs. However, this comes at the cost of some throughput.

3. Gencon — Uses concurrent marking along with a generational collec-

tor. It improves upon OptAvgPause, reduces fragmentation and still maintains better throughput. Useful for applications that use many short-lived objects.

4. Balanced — Concurrent marking is disabled. Uses region-based layout, i.e., heap is divided into small regions. These regions are individually managed to reduce maximum pause times and give efficient GC. This policy is optimized for throughput on large non-uniform memory access (NUMA) aware systems. Suitability of the balanced policy for this research is discussed in the subsequent sections.

3.4 Balanced Policy

This policy is intended for heap sizes larger than 4GB and is available only for 64-bit platforms.

3.4.1 Heap layout

As shown in Figure 3.3, the Java heap is split into thousands of equal sized regions. Age is tracked for each region in the Java heap, with 24 possible generations. Each region can be collected independently and has an age associated with it. An age 0 region, known as the eden space, contains the newest objects allocated. Objects surviving the GC cycles are gradually promoted to the region with higher age, i.e., old region. The highest age region represents a maximum age where all long-lived objects eventually reside.

3.4.2 Allocation Context

OMR provides support for non-uniform memory access (NUMA) architectures via an abstraction called an allocation context [20]. The allocation context represents the set of regions available to the heap region manager, which is responsible for any memory allocation.

3.4.3 Partial Garbage Collection

Partial garbage collection (PGC) [33] is triggered when the eden space is full. It is a stop-the-world event, i.e., the applications are stopped during PGC. As the name suggests, the set of memory regions selected for GC represents only a portion of the entire heap. The eden regions used for allocations since the last PGC are always included. Non-eden regions are selected based on the opportunity to reclaim more memory. The PGC favors Copy-Forward but can switch to Mark-Compact if the heap is too full. Since it examines only a subset of regions, some of the garbage may not be reclaimed. This is tackled in the global mark phase, explained next.

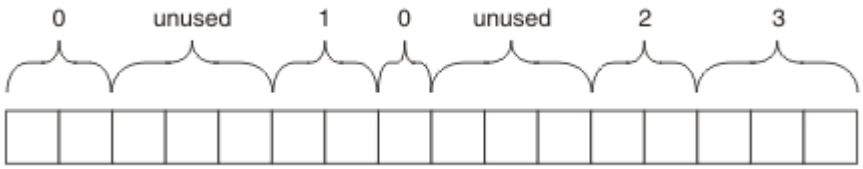

Figure 3.3: Heap Layout [1]

3.4.4 Global Mark Phase

A Global Mark Phase (GMP) [33] performs a mark operation on the entire Java heap, which allows it to see more dead objects than a PGC. A set of background threads is used for marking; they run concurrently with application threads, allowing better processor utilization. Objects that are traced by these threads may subsequently become unreachable by the time the collection process ends. Such unreachable objects that have not yet been reclaimed are referred to as floating garbage, which is bounded by the set of objects that died after starting a GMP. GMP updates live objects, i.e., a live-set for PGC, hence subsequent PGCs, can use a smaller live-set and do a more aggressive collection. A GMP live-set is cleared before the next global mark.

3.4.5 Global Garbage Collection

The Global Garbage Collection (GGC) runs mark-sweep on the entire heap and hence leads to longer pause times. The following situations can lead to a GGC cycle:

- A System.gc() call
- PGC collection rate cannot keep up with the memory demands.

If an application repeatedly triggers GGCs, it is considered an error in configuration.

3.5 Selecting a GC Policy for Cold Object Segregation

Segregation requires designating a separate memory area for the cold objects. The OptThruPut and OptAvgPause policies use an unsegmented heap and hence can not easily be used for segregation. This naturally leads to selecting region-based policies like Gencon and Balanced. The cold objects are likely to be promoted to the tenured regions. Hence, region age can serve as a factor to identify cold objects. Gencon uses only two generations. The balanced policy, on the other hand, offers 24 different generations and an opportunity to tweak the region selection for GC cycles. Additionally, the allocation context and per-object copying makes the balanced policy more suitable for this thesis. The JVM uses the Allocation Context (AC) structure to allocate objects into regions under the Balanced Garbage Collection (BGC) policy.

3.6 Cold Object Statistics

It is pertinent to look at the past research [34] that gathers cold object statistics as shown in Table 3.1. Access barriers (Section 4.2) are used to measure the last access time of the objects. Objects not accessed for a preset time threshold are treated as cold. The setup consists of 1024 regions having a size of 2 MB each. Various benchmarks show a varying number of cold regions. However, SPECjbb2005, a Java server benchmark [35] shows

Table 3.1: Cold region statistics.

Benchmark	Cold Regions	Ratio %
SPECjbb2005	143	13.96
SPECjvm2008 compiler.compiler	14	1.37
SPECjvm2008 compiler.sunflow	13	1.29
SPECjvm2008 derby	107	9.94
SPECjvm2008 sunflow	37	3.61
SPECjvm2008 xml.transform	42	4.10
SPECjvm2008 xml.validation	8	0.78

13.96 % cold regions. That means 13.96% regions do not have to do PGC frequently. Therefore, the GC pause time could be significantly reduced. Such cases assert the significant presence of cold objects and usefulness of pursuing this work.

3.7 Summary

A GC-cycle consists of mark-compact (Section 2.5.3) and copy-forward (Section 2.5.4) collections. Typically copy-forward collections are more than mark-compact operations. Hence, balanced policy copy-forward operation is modified to handle object segregation as part of this research. The segregation also requires separate regions, i.e. cold regions, which is provided by the cold allocation context [12]. The cold allocation context provides an easy to manage list of cold regions, backed by a separate memory device, e.g., NVRAM or an SSD partition, which is explained in the next chapter.

Chapter 4

Measuring Object Temperature

Section 3.1 has described an object's temperature as a function of its access frequency. The techniques used to find the object access frequency often suffer performance degradation. The following sections explore various approaches used for identifying cold objects and to segregate them.

4.1 Stack walking

As described in Section 2.2.6 the Java stack is used to store frames for method calls. A frame is created on each method invocation and released when the method returns. The topmost frame on the stack belongs to the current method. The stack frame consists of local variables, operands, a reference to the runtime constant pool, and frame data. The local variables may be primitive data types or object references. Inactive objects do not give

any hint of activity and hence are hard to detect. Objects in the current stack frame are highly likely to be accessed and hence can be treated as hot references. Others can be treated as candidates for cold objects. The stack walking technique [5] uses the following approach to identify hot references.

- A daemon thread runs in the background and periodically signals mutator threads to sample stack frames.
- The top frames represent currently active methods and hence are the source of active references.
- The mutator thread samples the stack frames, and stores active references in a thread-local buffer.
- The mutator thread decides a safe point to sample the stack frames.

This approach can miss potential hot objects during sampling intervals and is effective with leaf objects only. Also, the runtime overhead for walking mutator stacks offsets the gains, which makes it less attractive.

4.2 Access Barriers

An access barrier allows the JVM to intercept any read or write operation to an object reference. These access barriers can be configured to record the last access time for an object. Any object that remains inactive for a certain time threshold should be treated as cold. The GC cycle being a stop-the-world operation offers the best opportunity to analyze the object access

timestamps. An additional method call on every object access is clearly not performant and hence this approach is only useful as a verification tool. The JIT mode imposes further restrictions. Frequently executed methods are compiled to native instructions by the JIT compiler, which takes away the majority of access barriers, limiting the usefulness of this approach i.e., it works only in JIT-off mode.

4.3 Page Protection

Operating system paging offers an alternate barrier approach by allowing the user to enforce page-level access permissions. The mprotect system call [36] provides an option to set the following permission bits for a heap range.

- PROT_NONE — The memory cannot be accessed at all.
- PROT_READ — The memory can be read.
- PROT_WRITE — The memory can be modified.
- PROT_EXEC — The memory can be executed.

Any violation of the access policy leads to a page fault. The user can register a customized fault handler to remember the access violation. The following steps briefly outline this approach:

- The whole heap is protected with PROT_NONE bits.
- A bitmask is maintained to represent each page of the heap memory.

- An object access invokes a fault handler that sets the bit for the containing page.
- The page protection is removed before the start of the GC cycle.
- A set bit indicates the presence of hot objects since the last GC cycle.
- The bitmask is used to segregate an object to a hot or a cold region.
- At the end of the GC cycle, the page protection is re-enabled.

This technique has less overhead compared to read/write access barriers discussed in the previous section. However, a single read/write access marks all the objects in the page as hot objects, creating false positives.

4.4 Card Tables

In addition to performance overhead, page protection is too coarse-grained to avoid false positives. This leads to also evaluate card tables [37] for tracking the reference updates. A card represents a contiguous chunk of memory in the heap, which is much smaller than a page size. A card table is a byte array, with one byte per card. The pointer stores are recorded by interpreting a zero byte as a dirty entry and a clean entry being a non-zero byte. Generational collectors use a card table to record the interesting pointers i.e., intergenerational references from old to new objects. This makes the card updates freely available for analyzing the object activity. Though it improves

the granularity over the page protection scheme, only write operations are recorded. Real life applications exhibit significantly higher read operations in comparison to writes, which limits the usefulness of card tables.

4.5 Application Profiling

Objects and class-level statistics are gathered at runtime over multiple application runs. This data is used in subsequent application runs to guide the object segregation. Past research around object cache locality [38] and field access frequency [39] gives an insight into the following properties relevant for this thesis:

- Fan-In — Number of incoming references to an object.
- Fan-Out — Number of outgoing references from an object.
- Temperature — How often this object is referenced by other objects.
- Popularity — How often an object references its children.

Temperature is calculated by summing up the access frequencies for each incoming reference. Similarly, the sum of frequencies for the outgoing references gives the popularity measure. Higher values of temperature and popularity indicate hot objects and conversely cold. Although the data is collected for individual objects, the final statistics reflect class-level properties. However, the applications may not exhibit similar characteristics during each run.

4.6 Summary

The techniques discussed earlier suffer the following issues.

- Measuring the object temperature at runtime incurs performance overhead.
- Page protection, in Section 4.3, is too coarse-grained to avoid false positives.
- Card tables, in Section 4.4, have limited usability due to write-only tracking.

This leads us to consider application profiling as an alternative, which clearly avoids the runtime overhead observed in the dynamic evaluation of object temperature. The next chapter discusses the application profiling implementation, evaluation, and future roadmap.

Chapter 5

Estimating Object Temperature with Application Profiling

5.1 Background

Application profiling measures the object temperature during an initial execution of a given application. This information is used in subsequent runs of the same application to segregate the objects into hot or cold regions. The objects do not persist across the application runs and hence the temperature data is summarized as class-level statistics. These statistics are gathered during the pre-collect phase of a GC cycle, i.e., when the application threads are stopped. This limits the ability to capture object access frequency. Hence,

the properties temperature and popularity mentioned in earlier chapter are redefined as follows.

- Temperature — How many other objects refer to objects of this class.
- Popularity — How many other objects are referred to by objects of this class.

The Java objects can be represented as an in-memory graph. A similar abstraction applies to classes as well. The classes are treated as graph nodes and any incoming/outgoing references define the direction of the edge. The cardinality of these incoming/outgoing references gives the weight of the edge.

A sample graph for a DaCapo run is shown for the 'java/lang/ref/SoftReference' class in Figure 5.1. The temperature (i.e., 96) is the sum of the weights for incoming edges. Similarly, the popularity (i.e., 102) is the sum of weights for all the outgoing edges.
The complete application-profiling solution works in the following three steps.

- Profiler — Collects class level statistics during a sample application run.
- Parser — Processes the statistics collected in the earlier step to classify the classes as hot or cold.
- Segregation during GC — Use the class level temperature information to segregate the objects into hot or cold regions during GC.

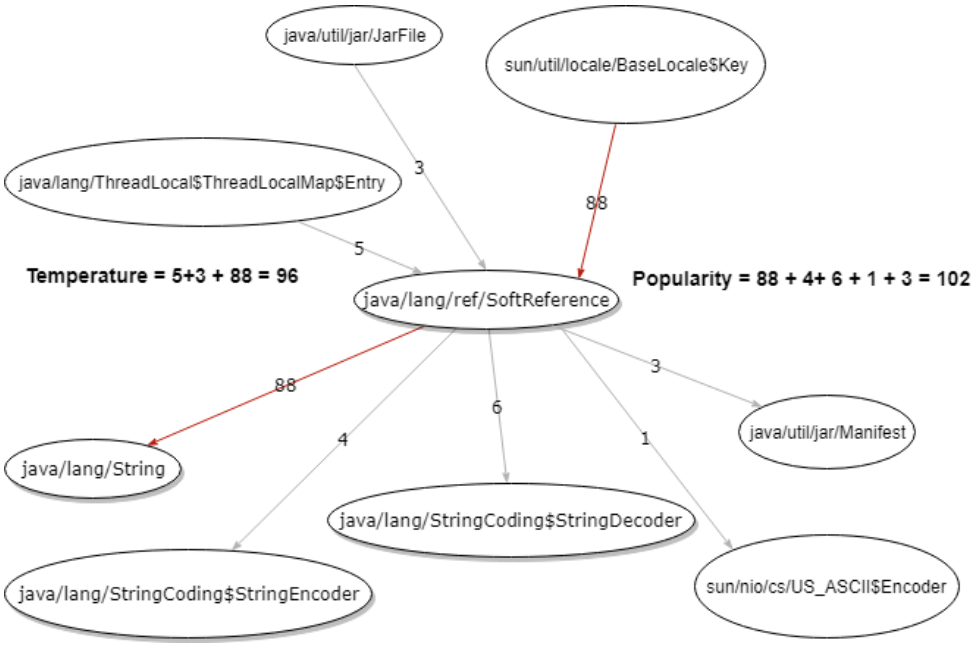

Figure 5.1: Class graph of java/lang/ref/SoftReference during first PGC

Detailed descriptions for each step are given below.

5.2 Profiler

The profiler collects class-level statistics across sample application runs, which are utilized in subsequent application runs. Application profiling is done during the pre-collect phase of the GC cycle. The mutators are stopped during this stage, which offers a safe point for object inspection. It consists of walking the entire heap to find class-level information as depicted in Figure 5.1

5.2.1 Hash Entry

The profiler gathers data in an in-memory hash table. A hash-table entry is depicted in Figure 5.2.

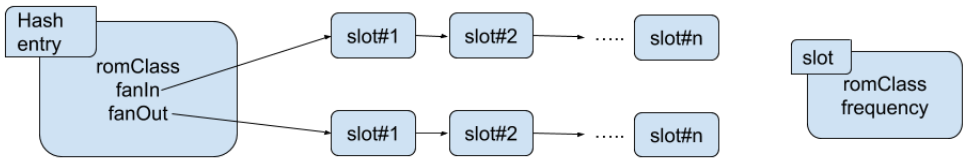

Figure 5.2: Profiler data structure

The hash entry represents a single class, which consists of the following fields:

- ROMClass — It is used to fetch the class name, which serves as a key for a hash table entry.

- fanIn list — A list of all the classes representing incoming edges as shown in Figure 5.1.
- fanOut list — A list of all the classes representing outgoing edges as shown in Figure 5.1.

5.2.2 The slot

The slot represents a reference field of an object for the fanOut list. Null references do not contribute to the fanOut list. In the case of the fanIn list, the slot represents an active referrer. The class-specific information for the slot is captured using the following fields:

- ROMClass — Similar to the hash entry, it identifies the class name for the reference field.
- frequency — Number of objects of this reference type forming an incoming/outgoing edge.

The summation of all the frequencies for the 'fanIn' list gives the class-temperature. Similarly, the summation of all the frequencies for the 'fanOut' list gives the class popularity.

5.2.3 Basic outline

Algorithm 1 shows the basic profiler steps.

Algorithm 1 The Profiler

```
1: Create hashTable
2: for each: object in Heap
3:     call profileInstance(object, hashTable)
4: textFile ← hashTable;
```

An in-memory hash table is created as a part of JVM initialization. All the heap objects are traversed and analyzed to find the temperature data. At the end of every Partial Garbage Collection (PGC) cycle, all the data is saved in a separate text file.

5.2.4 Profiling individual instances

The profileInstance method is described in Algorithm 2. It iterates over all the object slots and populates an in-memory hash table.

Algorithm 2 The Profiler Helpers.

```
1: procedure PROFILEINSTANCE(object, hashTable)
2:     // object: current object being traversed
3:     // hashTable: Figure 5.2
4:
5:     romClass ← object.romClass
6:     if (romClass ∉ hashTable) then
7:         hashTable ← romClass
8:     for each: slot in object
9:       invoke callback checkSlot(object, hashTable)
```

OMR provides an API that iterates over all the object slots and invokes a callback. The callback takes the following key arguments.

- Object — The object for which slots are iterated.

- User-defined data — The hash table is passed for this work.

The callback, i.e., the checkSlot method, has access to both the object and the current slot being iterated. This allows exploring object and slot relationships.

5.2.5 The callback

The checkSlot method described in Algorithm 3 updates frequency counts for both fanIn and fanOut linked lists. This method adds the slot object to the hash table and updates the fanOut list for the object being traversed. Similarly, the fanIn list of the slot is updated with an object entry. The fanIn and fanOut frequencies are incremented if the corresponding references are already present in the lists.

5.3 Categorizing Class Temperature

Every PGC cycle generates an output file as shown in Table 5.1. For example, the DaCapo benchmark generated 29 class data files. An entry for the class generally appears in multiple files with different values for temperature and popularity. The parser generates a hash table with a key-value pair being the class name and a tuple describing temperature information. The highest values are picked for the fanIn, fanOut, temperature, and popularity so as to capture object hotness during any of the runs.

Table 5.1: Class data during PGC.

fanIn	fanOut	temperature	popularity	Class Name
0	1	0	1	jdk/internal/module/ ModulePatcher
6	0	6	0	java/lang/ ClassLoader$AssertionLock
2	2	2	2	org/apache/commons/cli/ CommandLine
0	0	0	0	jdk/internal/jimage/ ImageReaderFactory$1
.	.	.	.	.
.	.	.	.	.
1	4	42	411	sun/net/www/protocol/file/ FileURLConnection
1	1	37	244	java/util/ ImmutableCollections$SetN$1
1	5	39	117	sun/util/locale/ LocaleObjectCache$CacheEntry
1	5	836	252	jdk/internal/loader/ BuiltinClassLoader$LoadedModule
.	.	.	.	.
.	.	.	.	.

Algorithm 3 The callback.

```
1: procedure CHECKSLOT(object, slot, hashTable)
2:     // slot: current object being traversed
3:     // object: containing object for the slot
4:     // hashTable: Figure 5.2
5:
6:     romClass ← object.romClass
7:     refRomClass ← slot.romClass
8:     if (refRomClass ∉ hashTable) then
9:         hashTable ← refRomClass
10:    if (refomClass ∉ object.fanOut List) then
11:        object.fanOut List ← refRomClass
12:    update refRomClass frequency
13:    if (romClass ∉ refRomClass.fanIn List) then
14:        slot.fanIn List ← romClass
15:    update romClass frequency
```

The parser sorts all the class summary data based on temperature first and using popularity as a secondary key. The classes with higher values for temperature and popularity are treated as hot classes, cold otherwise. The final output is a file with a list of cold classes, which is used in subsequent runs as explained in the next section.

5.4 Temperature Based Segregation

The PGC consists of either a copy-forward or mark-compact collection. It allows copying an individual object into a specific region, e.g., older objects are moved to the tenured region. The temperature data is used during this phase to further segregate objects into hot or cold regions, which consists of

the following steps.

- The profiled data is read into an in-memory hash table during JVM initialization.
- A page-fault handler is registered to use OS paging as an access barrier.
- As detailed in Section 4.3, the page protection technique identifies hot pages.
- A single page fault can mark all the objects in a page as hot. The profiled data is used to refine the object classification as hot or cold.

5.4.1 Copy-forward collection

The copy-forward collection is pre-dominant during PGC cycles, which is modified as detailed in Algorithm 4. It augments the page protection technique detailed in Section 4.3 with the following two changes.

- Only non-Eden regions are protected
- Profiled data further guides the final destination for the object.

5.4.2 Performance concern

The hash table needs to be queried for the temperature data. It adds up the lookup overhead per object copy. In the production environment, this overhead could be easily avoided by encoding the temperature data in the

Algorithm 4 Copy Forward.

```
1: // hashTable: Figure 5.2
2: // object: object being copied during copy-forward collection;
3: region ← region containing object
4: if (region = nonEden) then
5:     page ← page containing object
6:     if (page = hot) then
7:         profiledTemp ← object temperature in hashTable
8:         if (profiledTemp = hot) then
9:             move object to hot region
10:        else
11:            move object to cold region
12:    else
13:        move object to cold region
```

class header. The more elaborate information about the experimental setup and results for the proposed solution is given in the next chapter.

5.5 Summary

This chapter can be best summarised by listing all the contributions made as a part of this thesis as follows.

- The profiler (Section 5.2) generates temperature data for all Java classes of an application.
- The parser (Section 5.3) analyses the temperature data and generates list of cold classes.
- The list of cold classes is further utilised by the copy-forward collection (Section 5.4.1) for object segregation.

- The page-protection is used in controlled manner, i.e., only non-Eden regions are protected. This is done to reduce the overhead of page-faults and also due to higher likelihood of finding cold objects in older regions.

This also gives us guidance to test possible configurations, which are explained further in the next chapter (Section 6.2).

Chapter 6

Experimental Setup and Results

This chapter discusses the experiments that were performed and the corresponding results.

6.1 Build and Test Setup

All the experiments are conducted inside a Docker container [40] with the following configuration.

- Host Memory — 16GB
- Host Swap space — 16GB partition
- Cold storage — 16GB SSD partition

- Docker image — Ubuntu 16.04.3 LTS
- Kernel Version — 3.13.0-129-generic
- CPU — Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz

The Docker container uses memory and swap space of the host operating system. The cold storage is a partition on the host OS exported to the Docker container. The Docker overhead is negligible since the host OS is sharing the kernel with the running Docker container. All the testing is done with a non-debug OpenJ9 build for x86-64 platform.

6.2 Test Configurations

The final solution consists of page protection used only for non-eden regions and profiled data used to segregate the objects. The cold regions are backed by an SSD partition. It is tested against four different configurations described below.

1. Unmodified JVM [unmodified]
2. The entire heap is page protected. [protectAll]
3. Only tenured regions are page protected. [noEden]
4. Application profiling is used along with page protection applied to the tenured regions. [Profiled]

Subsequent sections show the benchmark results for all four configurations.

6.3 DaCapo Benchmarks

Details about the DaCapo benchmarks are given in Section 2.6.1. Benchmarks in the DaCapo suite not using the OpenJDK classes were removed from the experiments. Each of the remaining benchmarks, i.e., avrora, fop, h2, jython, luindex, lusearch, pmd, sunflow and xalan were run 60 times. The initial 10 runs were discarded to account for application warm-up. These runs use the *perf* utility [41] for all four configurations enumerated above.

The results are depicted as Box plots, which use data for all 50 warm runs per benchmark. Initially, an individual dataset is sorted and then divided into four groups. Each of the groups contains 25% of the data points. The lines dividing the groups are called quartiles, and the groups are referred to as quartile groups. The median (middle quartile) marks the mid-point of the data and is shown by the line that divides the box into two quartile groups. The box in the middle consists of 50% of the data points. The remaining two quartile groups consist of data points between the box and the bar. The results show the average runtime in milliseconds. Lower values imply better performance.

The final solution suffers due to some of the following reasons.

- Profiled data is stored as a hashmap, which requires lookup during every segregation operation. This overhead can be eliminated by keeping temperature data in class headers.

- GC cycle typically consists of copy-forward and mark-compact oper-

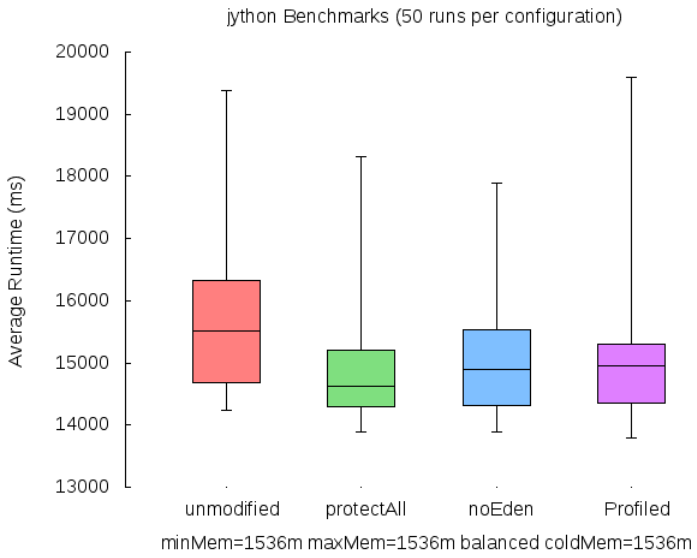

Figure 6.1: Jython benchmark results

ations. Copy-forward was modified to handle the object segregation, which dominates over mark-compact. Any application with fewer copy-forwards performs poor segregation.

Jython (Figure 6.1) shows good performance over the unmodified JVM for page marking. Jython is an interpreter, which shows very similar, highly regular behavior across various workloads, suggesting that the interpreter rather than the interpreted program dominates. The predictability is more favorable for paging solutions and even for the static analysis. Although the profiled data adds some overhead, results are still better than the unmodified JVM. The profiled solution shows more cache references while the cache-miss rate came down by 2% compared to the unmodified JVM.

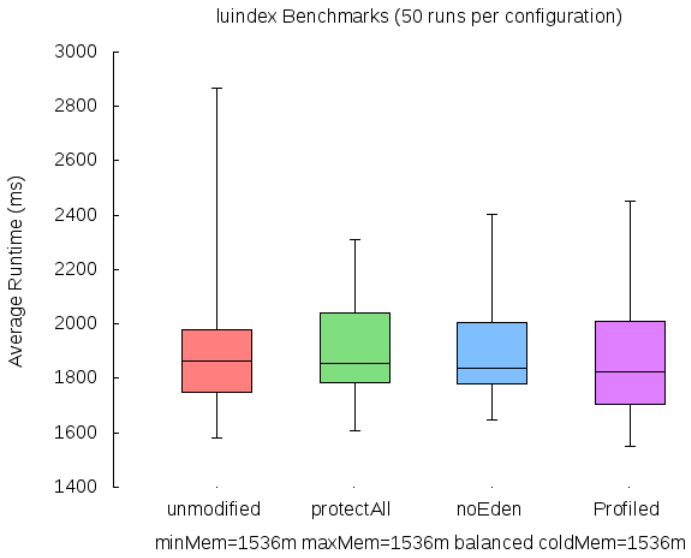

Figure 6.2: Luindex benchmark results

The luindex benchmark (Figure 6.2) has a nursery survival rate [14] of 23.7%, which indicates substantial GC workload. This benchmark is building an index, so many objects are long-lived, rarely to be accessed again, i.e., more cold objects. Furthermore, the final solution shows more cache references while the cache miss rate stays the same compared to the unmodified JVM. The runtime gradually improves, suggesting a good candidate for application profiling and page-marking approach.

Avrora (Figure 6.3) results do not show any improvement over the unmodified JVM. However, protecting only non-eden regions gives clear gains due to reduced page faults. Also, the profiling improved segregation. The results suggest using a better alternative to page marking will improve the runtime

for such applications.

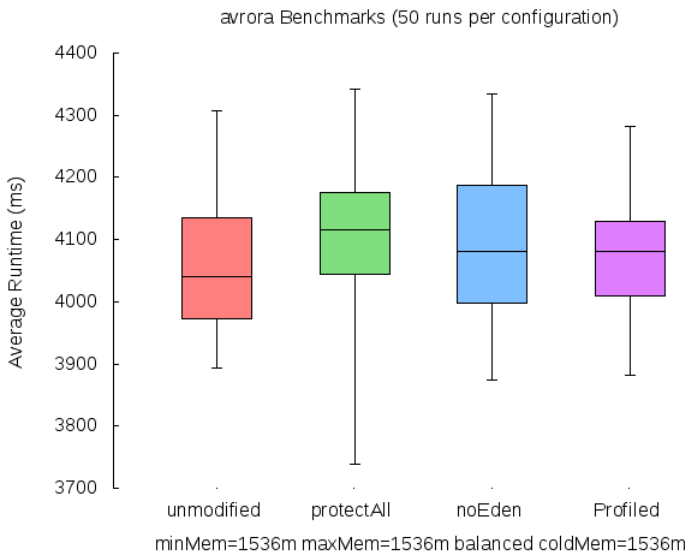

Figure 6.3: Avrora benchmark results

Fop (Figure 6.4) and pmd (Figure 6.5) have a nursery survival rate of 14% and these benchmarks test the ability for a JVM to perform a calculation on an input set. This leads to fewer long-lived objects and hence page protection performs poorly. Non-eden-only page protection helped reduce page faults and application profiling improved segregation. However, these gains barely improve over the unmodified JVM. Better alternatives should be explored to be combined with application profiling to address this issue.

H2 (Figure 6.6) is an in-memory database benchmark with the highest nursery survival rate of 63.4%, which should expect speedup due to temperature-based segregation. Investigation of the GC logs shows that the JVM was

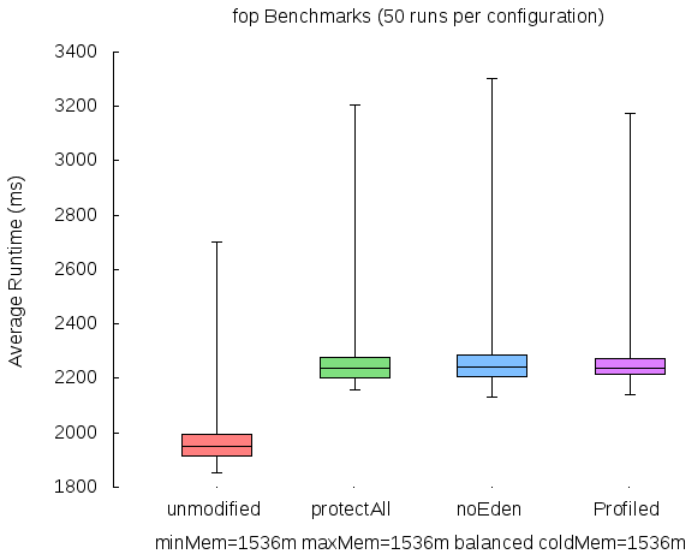

Figure 6.4: Fop benchmark results

unable to perform a copy-forward operation due to the heap being too full. Hence not enough segregation was performed for the H2 benchmark, leading to poor run time with page protection. Incorporating the segregation logic for mark-compact operations should address this issue. Application profiling certainly improved the segregation and hence does slightly better over the non-eden-only page-protection solution.

Lusearch (Figure 6.7) and xalan (Figure 6.8) have a far lower nursery survival rate, i.e., less than 4%. Sunflow (Figure 6.9) also is computationally intensive and most of the objects die young. These benchmarks are not expected to perform well with a page-marking scheme, which is evident in the results. Non-eden-only protection does not give any measured gains over pro-

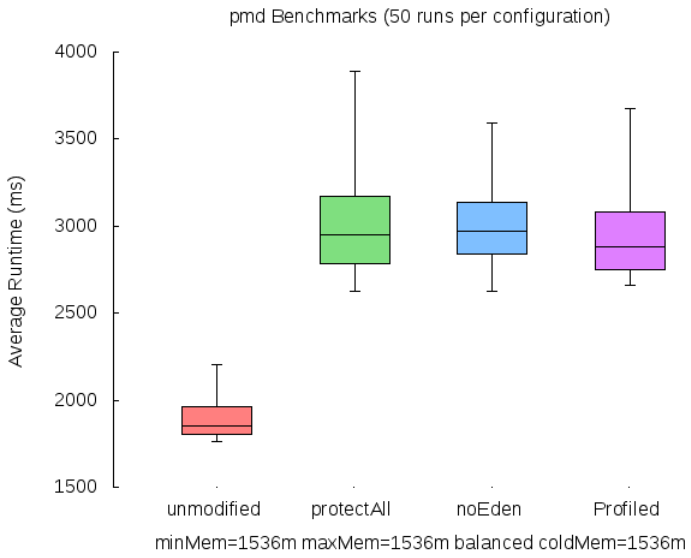

Figure 6.5: pmd benchmark results

tecting the entire heap. Even though the profiling marginally improved the performance, it is not worth exploring for such applications. Long-running applications with a better nursery survival rate will benefit more with this solution.

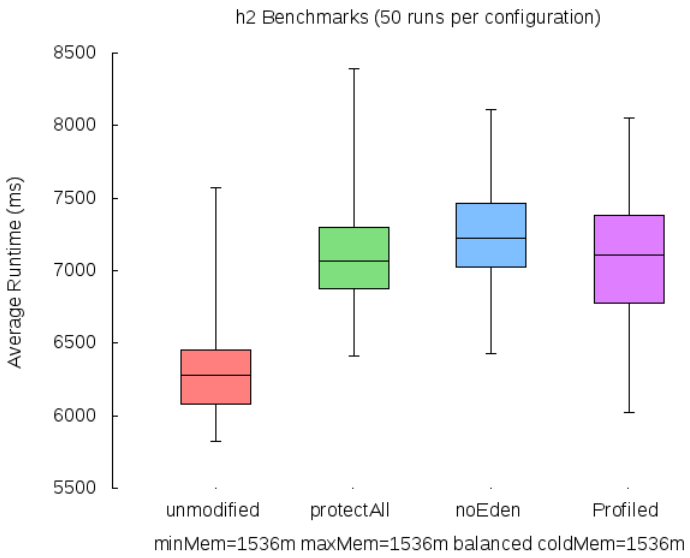

Figure 6.6: H2 benchmark results

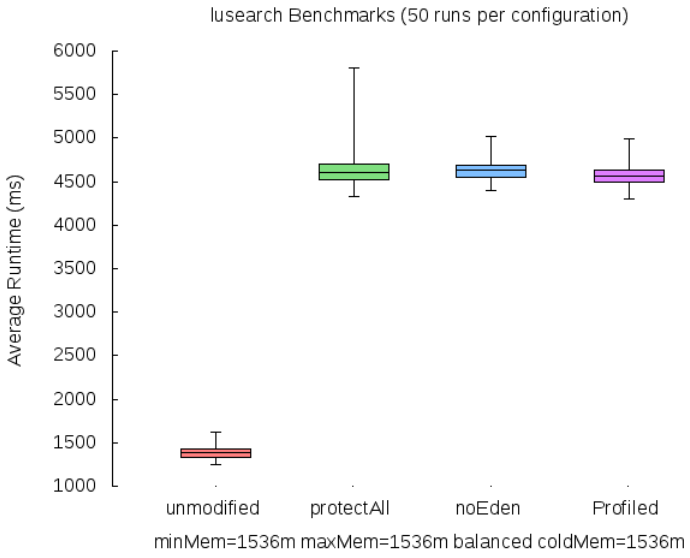

Figure 6.7: Lusearch benchmark results

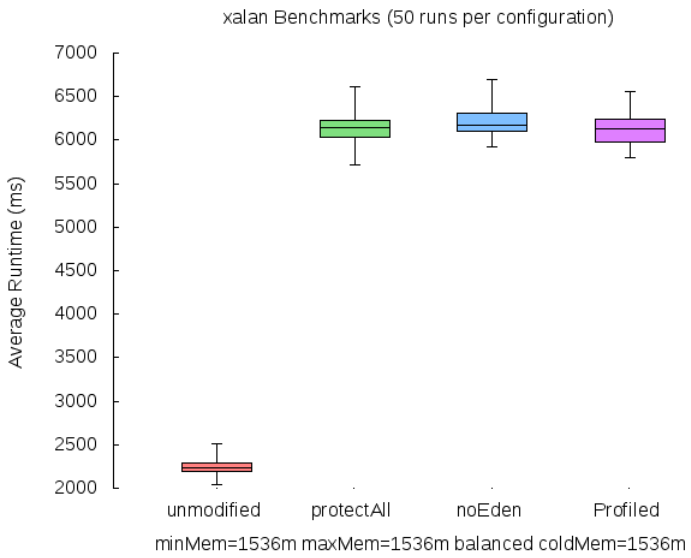

Figure 6.8: Xalan benchmark results

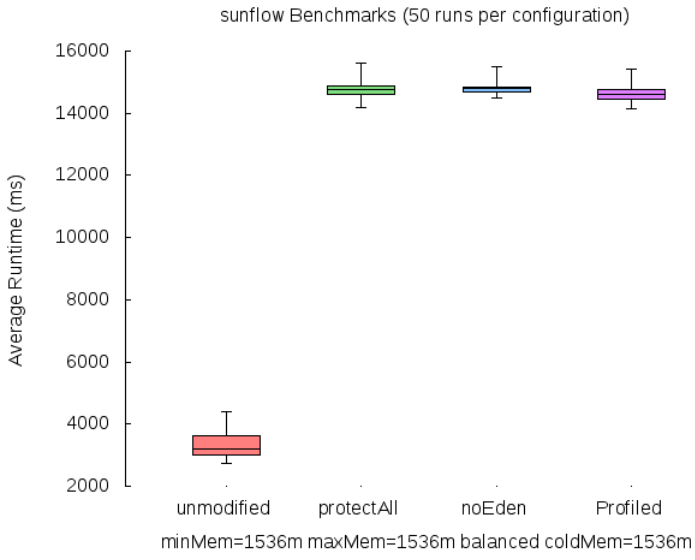

Figure 6.9: Sunflow benchmark results

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis explores the properties of cold objects and corresponding segregation techniques. The cold and hot objects co-exist in the heap, which affects cache performance and increases GC time. Segregating cold objects to different memory area solves the issue of cache pollution. This also leads to a smaller number of hot regions and reduced garbage collection frequency for cold regions. Furthermore, cold objects can be stored to a separate memory device allowing an application to use more hot heap. The current implementation uses an SSD partition as cold storage, which can be easily changed to use cheaper storage like NVRAM. This also solves the cache pollution issue and hence gives better performance. Finally, this work is concluded as follows.

- *Page protection* is used to identify hot pages. Protecting the entire heap generates more page faults, leading to poorer performance over the unmodified JVM. However, applications showing predictable behavior benefit from such a coarse-grained solution, e.g., Jython benchmark.

- *Cache-miss rate* Applications that showed improved performance with the application profiling also showed improved cache reference and lower cache-miss rate, e.g., luindex and jython.

- Protecting only *non-eden regions* certainly helped in the case of avrora and luindex, which generated fewer page faults. Additionally higher likelihood of finding cold objects in older regions (non-eden regions) than in the eden space helped with performance.

- Benchmarks that are *computationally intensive* and having low nursery survival rate, i.e., fewer cold objects perform badly, which is expected. For example, sunflow, lusearch and pmd do not show any noticeable changes in the results for the non-eden-only page protection.

- A page is marked hot on the first object access, leading to false positives, i.e., the remaining objects in the page are marked hot too. Although a runtime analysis can reduce these false positives, it will introduce more overhead. *Application profiling* can assist in such situation by providing an approximation of the class temperature. The benchmarks fop, pmd, h2 and lusearch show clear improvements using profiled data in comparison with the paging-only solution.

- *Classification is sub-optimal*, because the assumption that all objects of a given class have the same temperature is not correct.

Application profiling alone is not sufficient to tackle the temperature based object segregation. Runtime analysis is inevitable due to changing object access pattern across application runs. However, any such runtime evaluation should be done for a very short interval so as not to affect the performance. The results suggest application profiling either perform better or remain at the same level as the non-profiled tenured only solution. Luindex and Jython performed better than unmodified. A similar performance can be achieved with H2, which shows better segregation via profiling. Implementing segregation for a mark-compact operation will direct impact memory-intensive applications like H2. Furthermore, the ability to back the cold region with a cheaper memory provides a cost-effective solution to increase heap memory. Any long-running application requiring larger heap memory will surely benefit with such an approach.

7.2 Future Work

Application profiling keeps all the temperature data in an in-memory hashtable. A hashtable lookup adds unnecessary overhead during garbage collection (GC). This information could be easily encoded in object headers to save the lookup time.

Cold regions are currently backed by an SSD partition. Instead, all the

experiments should be conducted with cold regions backed by an NVRAM backed device to see actual gains.

The mark-compact operation of a GC cycle needs to be changed to handle object segregation, which will significantly improve the performance for memory-intensive applications like h2.

The Balanced GC selects a region for GC based on the best returns. The region selection needs to be enhanced based on region type, i.e., cold or hot. Hot regions should be prioritized over cold regions.

The application behavior changes across the runs. Hence, the profiled data should be re-evaluated at runtime. The IBM Z14 platform supports guarded storage facility (GSF) [42], which allows applications to guard a memory region. A hardware read barrier is provided for such guarded regions to give better granularity over page protection.

Stack walking (Section 4.1) can be used to update the temperature data at runtime. Considering only the JIT-compiled methods [43] can significantly reduce the runtime overhead associated with the stack walking.

Bibliography

[1] IBM. Region age balanced policy. `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_gc_balanced_region_age.html`, 2011. Accessed: 2019-06-17.

[2] IBM. Eclipse OpenJ9-JIT. `https://www.eclipse.org/openj9/docs/jit/`, 2019. Accessed: 2019-06-17.

[3] J. Gosling, B. Joy, et al. *The Java Language Specification Java SE 7 Edition*. Oracle America, Inc., 500 Oracle Parkway M/S 5op7, California 94065, U.S.A, 2011.

[4] A. W. Appel. Simple generational garbage collection and fast allocation. *Software–Practice & Experience*, 19(2):171–183, February 1989.

[5] K. Briggs, B. Zhou, and G. Dueck. Cold object identification in the Java virtual machine. *Software–Practice & Experience*, 47:79–95, 2017.

[6] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. *SIGARCH Comput. Archit. News*, 42(3):265–276, June 2014.

[7] Intel. Intel optane technology disruptive memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology-animation.html, 2019. Accessed: 2019-06-17.

[8] Thomas Mikolajick, Christine Dehm, Walter Hartner, Ivan Kasko, MJ Kastner, Nicolas Nagel, Manfred Moert, and Carlos Mazure. Feram technology for high density applications. *Microelectronics Reliability*, 41(7):947–950, 2001.

[9] R. Neale, D. Nelson, and Gordon Moore. Non-volatile and programmable, the read-mostly memory is here. *Electronics*, pages 56–60, 1970.

[10] S. Tehrani, B. Engel, et al. Recent developments in magnetic tunnel junction MRAM. *IEEE Transactions on Magnetics*, 36:2752–2757, 2000.

[11] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[12] Scott Young, Michael Flawn, Gerhard W. Dueck, Kenneth B. Kent, and Charlie Gracie. Persistent memory storage of cold regions in the openj9 java virtual machine. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, CASCON '18, pages 213–223, Riverton, NJ, USA, 2018. IBM Corp.

[13] T. Lindholm, F. Yellin, et al. *The Java Virtual Machine Specification Java SE 7 Edition.* Oracle America, Inc., 500 Oracle Parkway M/S 5op7, California 94065, U.S.A, 2011.

[14] S.M. Blackburn, R. Garner, C. Hoffman, 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. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct 2006. ACM Press.

[15] P. Patros, K.B. Kent, and M. Dawson. Investigating the effect of garbage collection on service level objectives of clouds. *Proceedings of the 19th IEEE International Conference of Cluster Computing*, pages 633 – 634, 2017.

[16] IBM. Eclipse OpenJ9. `https://github.com/eclipse/openj9`, 2018. Accessed: 2019-06-17.

[17] IBM. Heap lock allocation. `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_allocation_heaplock.html`. Accessed: 2019-06-17.

[18] IBM. Thread local heap. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_allocation_cache.html. Accessed: 2019-06-17.

[19] IBM. Portability library. https://www.eclipse.org/omr/reference/port.html, 2019. Accessed: 2019-06-17.

[20] IBM. Eclipse OMR. https://github.com/eclipse/omr, 2018. Accessed: 2019-06-17.

[21] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook*. CRC Press, 6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742, 2012.

[22] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[23] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[24] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 21–21, Berkeley, CA, USA, 2001. USENIX Association.

[25] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[26] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. pages 106–116, Copenhagen, Denmark, June 1993.

[27] S.M. Blackburn, R. Garner, C. Hoffman, 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. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java benchmarking development and analysis (extended version). Technical Report TR-CS-06-01, ANU, 2006. http://www.dacapobench.org.

[28] open source. JDBCBench:profiling database/jdbc-driver. `https://github.com/openlink/jdbc-bench`, 2019. Accessed: 2019-07-04.

[29] Apache. The apache lucene project develops open-source search software. `https://lucene.apache.org/`, 2019. Accessed: 2019-07-04.

[30] Daniel A. Menascé. Tpc-w: A benchmark for e-commerce. *IEEE Internet Computing*, 6(3):83–87, May 2002.

[31] Paul R. Wilson and Thomas G. Moher. A 'card-marking' scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Notices*, 24:87–92, 1989.

[32] Stephen M Blackburn and Kathryn S. McKinley. In or out?: Putting write barriers in their place. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 175–184, New York, NY, USA, 2002. ACM.

[33] IBM. Balanced garbage collection policy. `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_8.0.0/com.ibm.java.vm.80.doc/docs/mm_gc_balanced.html`. Accessed: 2019-06-17.

[34] B. Zhou. Cold objects in the Java virtual machine. Master's thesis, University of New Brunswick, 3 Bailey Dr, Fredericton, NB E3B 5A3, 2015.

[35] Standard Performance Evaluation Corporation. SPECjbb2005. `https://www.spec.org/jbb2005/docs/WhitePaper.html`, 2005. Accessed: 2019-06-17.

[36] Linux Programmer's Manual. mprotect, pkey_mprotect - set protection on a region of memory. `http://man7.org/linux/man-pages/man2/mprotect.2.html`, 2019. Accessed: 2019-06-17.

[37] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. 1999.

[38] M. Dombrowski, K. Nasartschuk, K. B. Kent, and G. W. Dueck. A survey on object cache locality in automated memory management systems.

In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 349–354, May 2015.

[39] Taees Eimouri, Kenneth B Kent, and Aleksandar Micic. Effects of false sharing and locality on object layout optimization for multi-threaded applications. In *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 1–5. IEEE, 2016.

[40] Docker. Docker container. `https://www.docker.com/`. Accessed: 2019-06-17.

[41] M. Kerrisk. Perf manual. `http://man7.org/linux/man-pages/man1/perf.1.html`. Accessed: 2019-06-17.

[42] IBM. How concurrent scavenge using the guarded storage facility works. `https://developer.ibm.com/javasdk/2017/09/25/concurrent-scavenge-using-guarded-storage-facility-works/`, 2017. Accessed: 2019-06-17.

[43] Michael R Jantz, Forrest J Robinson, Prasad A Kulkarni, and Kshitij A Doshi. Cross-layer memory management for managed language applications. In *ACM SIGPLAN Notices*, volume 50, pages 488–504. ACM, 2015.

Vita

Candidate's full name: Abhijit Shantaram Taware
University attended (with dates and degrees obtained):

- Pune University 1999-2003 B.E. Information Technology with first class
- University of New Brunswick 2017-2019 MCS

Publications:

- S. Rudrapatna, N. Narendra Kumar, A. Taware and P. Vyas, 'An Experimental Flow Secure File System,' 2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), New York, NY, 2018, pp. 790-799.

Conference Presentations:

- A. Taware, K. B. Kent, G. W. Dueck, and C. Gracie, 'Cold Object Identication, Sequestration and Revitalization,' CASCON'18 Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering, Toronto, Poster, October 2018.
- A. Taware, K. B. Kent, G. W. Dueck, and C. Gracie, 'Garbage Collection of Cold Regions,' UNB Research Expo, Fredericton, Poster, April 2018.