DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

An Experimental Study on the Behavioural Tendencies of Objects Classified As Hot and Cold by a Java Virtual Machine Garbage Collector

HANNA NYBLOM

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

An Experimental Study on the Behavioural Tendencies of Objects Classified As Hot and Cold by a Java Virtual Machine Garbage Collector

HANNA NYBLOM

Master in Computer Science
Date: May 27, 2020
Supervisor: Philipp Haller
Examiner: Johan Håstad
School of Electrical Engineering and Computer Science
Host company: Oracle Svenska AB
Swedish title: En experimentell studie av trender i beteende bland objekt klassificerade som heta och kalla av en skräpsamlare för *Java virtual machine*

Abstract

A constitutive hypothesis of the Java Virtual Machine garbage collector ”ThinGC”, presented by Mingkun Yang et al. [1], an extension of Oracle’s ”ZGC”, is that capitalising on possible temporal locality could optimise collection by limiting the total number of objects to manage. To achieve GC optimisations, ThinGC classifies objects as hot (recently referenced) or cold, separates the hot and cold objects into distinct memory spaces, and collects the spaces separately using two garbage collectors.

In order to examine to what extent this temporal locality can actually be observed, this thesis analyses the behaviour of objects classified as hot and cold by ThinGC. Reviewed behaviour includes: tendency of cold objects to remain cold, expected length of cold streaks, and if the tendency to remain hot or cold is related to the type of the object.

In order to examine object behaviour, hotness information for all objects in selected benchmarks from the DaCapo benchmark suite is logged in each GC cycle of ThinGC. The hotness information of each object is then compiled following address forwardings, and metrics estimating the behaviour of each object is calculated.

Analysis of the charts and tables presenting the results of the metric calculations show for instance that ”reheats” of objects are uncommon, cold objects usually stay steadily cold, and long cold streaks are more common than long hot streaks.

The results highlight distinctly different behaviours of hot and cold objects and indicate that the concept of classifying objects by hotness and treating cold objects separately could be well founded.

The results also show some classes of objects being more or less likely to stay cold. If these class behaviours could eventually be proven to be reliable by examining a larger set of programs, the information could be useful as a baseline for GC tuning. Also, if hotness information were collected and similar metrics were calculated concurrently, this information could aid in live GC decision making.

Sammanfattning

En grundläggande hypotes för Java virtual machine skräpsamlaren ”ThinGC”, presenterad av Mingkun Yang m. fl. [1], en utveckling av Oracles ”ZGC”, är att ett nyttjande av eventuell temporal lokalitet skulle kunna optimisera skräpsamling genom att begränsa den totala mängden objekt som behöver behandlas. För att optimisera skräpsamling klassificerar ThinGC objekt som varma (dvs. nyligen refererade) eller kalla, separerar heta och kalla objekt i skilda minnesutrymmen, och skräpsamlar dessa minnesutrymmen separat med två skräpsamlare.

För att undersöka i vilken utsträckning denna temporala lokalitet faktiskt kan observeras, granskar denna uppsats beteende av objekt klassificerade som heta eller kalla av ThinGC. Undersökta beteenden inkluderar: tendens bland kalla objekt att förbli kalla, antal cykler objekt förväntas förbli kalla och om tendenser bland objekt att förbli heta eller kalla är relaterade till objekttyp.

För att kunna granska objekts beteenden loggas värmeinformation för alla objekt i utvalda ”benchmarks” från benchmarksviten DaCapo under varje skräpsamlingscykel av ThinGC. Varje objekts värmeinformation kompileras sedan genom att följa vidarebefodringar av adresser, och till slut beräknas mätetal som uppskattar objektens beteende.

Analys av de diagram och tabeller som presenterar resultaten av beräkningen av mätetalen visar, till exempel, att återuppvärmningar av objekt är sällsynta, att kalla objekt oftast håller sig stadigt kalla, och att kalla objekt oftast håller sig kalla längre än varma objekt håller sig varma.

Resultaten visar tydligt skilda beteenden för heta och kalla objekt och indikerar att konceptet att klassificera objekt efter värmeinformation och behandla heta och kalla objekt separat kan vara välgrundat.

Resultaten visar också att vissa objektklasser är mer eller mindre benägna att hålla sig kalla. Om dessa klassbeteenden kan visas vara pålitliga genom att undersöka en större mängd program skulle informationen kunna vara användbar som en baslinje för finjustering av skräpsamling. Om värmeinformation kan samlas, och liknande mätetal beräknas parallellt med exekvering, skulle denna information också kunna bistå skräpsamlaren med att ta direkta beslut.

Acknowledgements

I would like to thank my supervisors, associate professor at KTH, Philipp Haller, associate professor at Uppsala Universitet, Tobias Wrigstad, and principal member of technical staff at Oracle, Jesper Wilhelmsson.

I would especially like to thank PhD student at Uppsala Universitet, Albert Mingkun Yang, who not only provided constant guidance, but also crucially and exceptionally introduced ZGC and ThinGC in detail.

Thank you all for your time, guidance, and for sharing your knowledge.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Problem . . . 2
- 1.3 Objective and Purpose . . . 2
- 1.4 Research Question . . . 2
- 1.5 Contributions . . . 3
- 1.6 Sustainability, Ethics and Societal Aspects . . . 3
- 1.7 Outline . . . 4

2 Background and Related Work **5**
- 2.1 Garbage Collection . . . 5
- 2.2 ZGC . . . 6
  - 2.2.1 ZGC Phases . . . 7
- 2.3 ThinGC . . . 9
- 2.4 Related Work . . . 10
  - 2.4.1 Object Behaviour . . . 10
  - 2.4.2 Object Hotness . . . 10
  - 2.4.3 Copying GCs and Locality of Reference . . . 11
  - 2.4.4 Generational GCs and Object Lifetime Prediction . . . 11
  - 2.4.5 Object Connectivity . . . 12

3 Methods **13**
- 3.1 Overview . . . 13
- 3.2 Implementation . . . 14
  - 3.2.1 Logging . . . 14
  - 3.2.2 Compiling Hotness Information . . . 16
  - 3.2.3 Calculating Hotness Metrics . . . 17
- 3.3 Reliability . . . 19
- 3.4 Application to Other Areas . . . 20

3.5 Limitations 20

4 Results **22**
4.1 Overview 22
4.2 Benchmark suite 23
4.3 Plots 28
4.3.1 Reheats 28
4.3.2 Ratios 32
4.3.3 Streaks 35
4.3.4 Main Temperature 38
4.4 Tables 42
4.4.1 Average Consecutive Streak 43
4.4.2 Mostly & consistently hot/cold objects 49
4.4.3 Reheats 55

5 Discussion and Conclusions **59**

Bibliography **61**

Chapter 1

Introduction

Since Java helped introduce garbage collection (GC) to mainstream computing, widening the use, garbage collectors has improved significantly. Garbage collection still introduces unwanted costs however. For applications with large amounts of traffic and growing heap sizes, inefficient garbage collection can introduce latency and increase the runtime of typical data processing tasks by up to 40% [2]. Consequently, the push towards the ideal "invisible" collector continues.

1.1 Background

Since the effectiveness of garbage collection is coupled with application performance, optimising garbage collection, previously through approaches like parallelisation, concurrency and generational partitioning, is an important objective of the GC field. Even garbage collection schemes which generate improved performance have their shortcomings. In the case of generational garbage collection these include fragmentation, tenured garbage and the cost of the generational write barrier are challenges [3].

A new garbage collection scheme introduced by Mingkun Yang et al. [1], "ThinGC", classifies objects as hot or cold depending on whether they were recently used by the program, separates hot and cold objects into distinct memory spaces (hot-storage and cold-storage) and collects these spaces separately using separate garbage collectors for hot and cold objects. The hypothesised benefits of this scheme are a reduction in the number of live objects managed concurrently by the hot-storage collector, and the possibility of storing cold objects in cheaper, higher capacity memory.

1.2 Problem

Garbage collection schemes involving hotness often designate hotness to paths, regions or methods rather than directly to objects [4, 5]. Noted studies where hot/cold data identification and separation have been utilised [6, 7, 8, 1] don't look into the behavioural tendencies of objects classified as hot or cold. Consequently, behaviour such as stability and streak length of objects classified as hot or cold has previously not been thoroughly examined.

1.3 Objective and Purpose

The objective of this thesis is to investigate and evaluate if distinct behavioural tendencies can be observed for objects classified as hot and cold.

The purpose of this thesis is to provide insight into object behaviour in order to support development of garbage collection schemes involving object hotness.

Another objective of this thesis is to examine the potential of the approach to garbage collection optimisation which partitions the set of objects by their hotness.

1.4 Research Question

In order to explore the potential of the ThinGC approach we imagined the environment where this approach would perform optimally and explored if we could observe these conditions within a benchmark suite. We wondered:

- *Can we observe distinguishable behavioural tendencies for hot and cold objects?*

Example of research questions regarding tendencies which we imagined would be beneficial to the approach were:

- *Do objects have a tendency to stay cold once they have become cold?*
- *Do cold streaks tend to be longer than hot streaks?*
- *Do objects tend to be predominantly hot or cold during their lifetime?*

Once we had answers to these questions we felt that we could consider the following question:

- *Is it possible to justify some overhead of a GC optimisation approach which identifies and treats hot and cold objects separately?*

1.5 Contributions

The main contributions of the study are summarised as follows:

- A method to gather and compile object hotness information.
- A method and suggested metrics to evaluate object hotness information.
- A systematic experimental evaluation.
    - Resulting plots and tables.
- Insights into how the collected hotness information could be leveraged for optimisations.

The results of the study showed that distinguishable behavioural tendencies for hot and cold objects, which would benefit the ThinGC approach were prevalent. Examples of observed tendencies were: cold objects usually staying cold and cold streaks usually lasting longer than hot streaks. The results also showed that objects which were not consistently hot had a tendency to be predominantly cold during their lifetime.

From these results, we gathered that the GC optimisation approach of classifying objects by hotness and treating cold objects separately could be justified.

1.6 Sustainability, Ethics and Societal Aspects

This thesis has societal impact in its attempt to contribute to the research field of garbage collection and offer support for implementations of garbage collectors, specifically, but possibly not exclusively, in the hotspot virtual machine of OpenJDK.

Open source software development has arguable societal impact, and raises several interesting ethical issues, such as the freedom and public benefaction it provides [9, 10, 11].

Any further societal impacts would depend on what applications the possible future GC optimisations are used to optimise.

Since increased execution time implies increased energy consumption [12], these possible future GC optimisations would also combat increased energy consumption and improve the sustainability of software [13]. This is especially important since garbage collection can be a big contributor to runtime [2].

This study has been conducted in collaboration with the Oracle Corporation which has a stated code of ethics and business conduct [14, 15].

1.7 Outline

The thesis is presented as follows:

Chapter 2, Background and Related Work. This chapter provides needed context by summarising basics of garbage collection and giving an overview of Oracle's ZGC collector and the ThinGC collector. The chapter also discusses previous work.

Chapter 3, Methods. This chapter outlines and details the methods used to gather and process object hotness information.

Chapter 4, Results. This chapter presents and discusses the resulting plots and tables of the experimental evaluation.

Chapter 5, Discussion and conclusions. This chapter summarises and discusses the consequence and significance of the results. The chapter also considers possible future use of the results.

Chapter 2

Background and Related Work

2.1 Garbage Collection

Garbage collection [16] automatically manages memory by striving to reclaim memory occupied by objects that are no longer referenced by the program. Automatic memory management can replace manual memory management where memory needs to be manually and explicitly allocated and freed. Many common errors accompany failed manual memory management, such as memory leaks and invalid or uninitialised memory accesses to name a few. Avoiding these errors is one of the advantages of automatic memory management, along with avoiding inter-module dependencies and enabling fully modular programming [17]. A disadvantage however is the added overhead introducing latency which can be crucial to some applications. [18]

A tracing garbage collector traverses the tree of objects interconnected by references and identifies reachable objects in the graph. These objects are considered "live" while all other objects are considered "dead" [17].

The roots, or root set of the object tree are always reachable objects such as thread objects of currently running threads, objects currently on the call stack and classes loaded by the bootstrap or system class loader.

Other strategies for automatic memory management are reference counting and escape analysis. In reference counting, counts of the number of references to each object are maintained and garbage objects can be identified as objects with a zero reference count. Escape analysis evaluates whether the lifetime of a pointer can be proven to be restricted only to the current procedure and/or thread. When discussing garbage collection in this paper we will be referring to tracing garbage collection.

Some important metrics in garbage collection are pause time and through-

put. The pause time of a collection algorithm is the time in which the collector needs to halt program execution in order to ensure the integrity of the object trees. Throughput is the percentage of total execution time not spent in garbage collection.

The original tracing method for garbage collection is the "naive mark and sweep". This method consists of a mark stage and a sweep stage. In the mark stage the entire root set is traversed, marking every root object as live, every object being pointed to by a root object as live, continuing with the objects pointed to by this object. In the sweep stage, memory allocated by objects not marked as live is freed [17].

Variations on this original method include parallel, concurrent, compacting and generational collection. Parallel collection utilises multiple CPUs, if available, and more physical memory to run the same collecting algorithm on several threads, resulting in a faster collection. In concurrent collection one or more garbage collection tasks can be executed simultaneously with the application, which lowers pause time. Compacting collection strives to avoid fragmentation in memory by compacting all live objects and completely reclaiming the remaining memory. This enables the use of a single pointer for allocation, however a non compacting collector completes garbage collection faster. In a Generational collector, memory is divided into separate pools holding objects of different ages. Different algorithms are applied in the generations based on commonly observed characteristics. Generational collection exploit the following observations, known as the weak generational hypothesis: *most allocated objects die young and few references from old to young objects exist* [19, 20, 21, 22, 23].

In most of these variations, pause time is connected to heap size and processing large heaps would lead to longer pause times and more latency.

Since increased execution time implies increased energy consumption [12], decreasing energy consumption is a motivation for improving garbage collection.

2.2 ZGC

The Z Garbage Collector, ZGC, is the latest, currently experimental garbage collector from Oracle. ZGC is a mark compact, single generational parallel and concurrent collector. ZGC is designed to make pause times independent of heap or live-set size, and to never have pause times exceeding 10ms. Because the pause times are independent of the heap size, ZGC can efficiently handle heaps ranging from a few hundred megabytes to multi terabytes in size.

Being able to manage large amounts of memory without suffering application performance is important, for example, when serving a large amount of users concurrently [24].

One ZGC cycle consists of three main phases and three pauses where execution of the program halts. The main phases are the mark/remap phase (M/R), the selection of evacuation candidates phase (EC) and the relocation phase (RE). The three pauses are Stop The World 1 (STW1), Stop The World 2 (STW2) and Stop The World 3 (STW3). The phases will be presented further in 2.2.1.

In order to run concurrent garbage collection while presenting only valid pointers to the running program (the mutator) ZGC utilises two critical concepts: coloured pointers and load barriers. Coloured pointers uses four of the unused bits in the 64-bit pointer of 64-bit platforms to store information about the object. The four bits are named “Finalizable”, “Remapped”, “Marked0” and “Marked1”. The Finalizable bit (F) indicates if an object will be processed by a finalizer when collected. The Remapped (R), Marked0 (M0) and Marked1 (M1) bits are the three “colour” bits. At any given time during garbage collection, one of these three colours will be defined the “good” colour and the other two will be “bad” colours. The assignment of the good colour takes place in the STW1 and STW3 phases of ZGC, the first and third pause of a ZGC cycle . In the third pause, the R bit is always declared the good colour. In the first pause, either the M0 or M1 bit is declared the good colour. The good colour declared in the first pause alternates between M0 and M1 every other cycle.

A load barrier is code which is executed when a pointer from the heap is loaded. Load barriers are utilised by ZGC in order to ensure that any loaded pointer is replaced by the calculation of the corresponding pointer with good colour [24, 1].

2.2.1 ZGC Phases

STW1

The ZGC cycle starts with STW1. In STW1 the good colour is declared to be M0 or M1 and the roots of the object graph are added to the mark stack, which keeps track of objects which are yet to be marked. The roots are also passed through a mark barrier which detects and updates any invalid pointers to have the newly declared good colour.

M/R

The mark remap phase performs traditional recursive marking. The object graph is processed by GC threads using depth first traversal using a mark stack. A mark barrier changes the colour of any processed pointers. The liveness information on the page corresponding to the object is updated to correctly reflect the number of live bytes on a page. Marking of an object can fail in the case where multiple GC threads are marking objects in parallel and the object has already been marked. In this case, the pointer is still added to the mark stack to ensure full traversal of the graph.

A remap is performed by the mark barrier if the pointer points into the set of evacuation candidates.

STW2

STW2 is a synchronisation point. Here ZGC checks that the mark stacks are empty and the mark barriers are flushed from the mutators.

EC

In the evacuation candidates (EC) phase, the evacuation candidate set of sparsely populated pages is created. Evacuation candidates are pages which have not been allocated in the current GC cycle. The candidates with the least liveness information are added to the evacuation candidates set.

Pages in ZGC can be of small, medium or large size. Small pages contain small objects, medium pages contain medium sized objects and large pages contain only one large object. Because of this, the liveness information of a large page corresponds only to one object and large pages do not participate in relocation.

STW3

After the EC phase follows STW3 where the good colour is declared to be the R colour, the roots of the object graph are once again processed and are relocated if they point to any evacuation candidate, otherwise the pointer colour of the roots are changed to the new good colour R.

RE

In the subsequent relocation phase concurrent relocation is performed by several GC threads migrating all evacuation candidate pages. When the address

of an object is changed, this mapping from old to new address is recorded in the Forwarding table.

Mutator threads can help the GC threads with relocating the objects on evacuation pages if they access an object during this relocation phase, since the load barrier will recognise that the pointer does not have the good colour.

After all objects in all evacuation candidate pages have been relocated, the ZGC cycle is complete. Any pointers pointing to abandoned addresses are fixed either in the next mutator access or in the M/R phase of the next ZGC cycle [24, 1].

2.3 ThinGC

ThinGC is a collector which builds on ZGC and combines elements from generational GC and cache hierarchies. ThinGC aims to benefit from the principle of temporal locality by classifying objects as hot, if the object was recently accessed by a mutator, or cold otherwise. This classification is made possible by the existing coloured pointers and load barriers in ZGC. ThinGC strives to limit the number of live objects which needs to be managed by ZGC and to enable frequently accessed objects (hot objects) to be stored differently from rarely accessed objects (cold objects). Cold objects could for example be stored on cheaper, higher capacity memory. Reducing the number of live objects for ZGC to manage reduces the time of the mark/remap phase, which is the most time consuming phase of ZGC [1].

An object is marked as hot, either by the load barrier if it is accessed by a mutator in a ZGC cycle, or by the mark barrier in the M/R phase if the R bit in the object pointer is set to one, which means that the object has been accessed by a mutator during the time between two ZGC cycles.

Hot and cold objects are separated into hot storage and cold storage and managed by two different garbage collectors. Hot storage is managed by ZGC and cold storage is managed by the Cold Storage Garbage Collector (CSGC).

CSGC cycles are started by ThinGC after ZGC marking is done, unless there is already an active CSGC cycle, and runs on a dedicated thread. Pages with limited amounts of hot objects are moved from hot storage into cold storage by ThinGC.

Beyond reclaiming memory in cold storage, CSGC aims to identify unused ThinGC remembered set (remset) slots and remap pointers to reheated objects. CSGC, like ZGC, is a tracing garbage collector which starts with its root set, then follows references, but stops at the boundary between the heap and cold storage, marking corresponding remset slots as reachable and adding to the

roots of the following ZGC cycle. The root set of CSGC is populated by objects relocated from the heap to cold storage and fields of reheated objects which refers to objects in cold storage. In this way the root set of CSGC captures all pointers from the heap to cold storage.

Moving an object from hot storage into cold storage is called “freezing” and moving an object from cold storage into hot storage is called “reheating”. Objects in cold storage cannot be accessed directly by application threads. Before an access to a cold object is allowed to initiate, the cold object is reheated.

Validation against DaCapo benchmarks has shown that ThinGC did lower the amount of live objects to be managed by the hot storage collector [1].

2.4 Related Work

2.4.1 Object Behaviour

Dieckmann and Hölzle [25] measure the distribution of object types, as well as lifetimes, sizes and reference density (fraction of fields that contain pointers) for the SPECjvm98 benchmarks. Theirs is a nice complement to this work since it helps the reader understand the allocation behaviour of Java applications.

2.4.2 Object Hotness

Object Hotness Prediction

Seidl and Zorn [8] have proposed a method for dynamic storage allocation which can successfully predict which heap objects will be highly referenced (HR) at the time they are allocated. Predictions are produced by identifying HR objects based on training inputs, then using object information such as size, contents of call stack, the value of the stack pointer and the depth of the stack. They find that program references to heap objects are highly skewed and most objects are not highly referenced objects.

Akram et al. [26] explore write-intensity prediction based on object size, class type and allocation site in order to allocate highly mutated objects to the heap and rarely mutated objects in non-volatile memory. The exploration showed that write-intensity predictions of Java objects using allocation site are the most accurate.

Hot/Cold Classification in Flash Memory

Examples of successful implementations of hot/cold data identification and separation can be found in performance optimisation of flash memory, where it has great importance. Hot/cold data identification and separation has been used to improve the efficiency of garbage collection/cleaning policies [27, 28, 29, 30, 31], to optimise the flash translation layer [32], and in the design of SSD file systems [33].

These studies focus mainly on correctly identifying objects as hot or cold based on references and do not consider object behaviour once objects are classified as hot/cold.

2.4.3 Copying GCs and Locality of Reference

Previous work towards optimising copying garbage collectors usually designate hotness to regions or paths through a program, rather than objects, by identifying program traversal patterns and frequently executed methods. The use of traversal algorithms have been shown to improve locality of reference in the collected heap [34, 35, 36, 37]. No examination of hot and cold object behavioural tendencies were realised in these papers.

Chilimbi and Larus [38] proposed an online profiling scheme for a copying algorithm which observes objects accesses, constructs an object temporal affinity graph and uses the GC to rearrange the objects for better cache locality. They mention that most objects are small, often less than 32 bytes, and that most objects accesses are not lightweight (i.e. multiple fields are accessed together or an access involves a method invocation). Because of this, they argue that profiling can be implemented at object, not field granularity since most objects are smaller than cache blocks and their profiling instrumentation (several instructions per object access) will not incur a large overhead.

2.4.4 Generational GCs and Object Lifetime Prediction

Object lifetime prediction based on program allocation site has been studied with the aim of optimising generational garbage collection [39, 40, 41, 42, 43]. These schemes base their predictions solely on allocation site and do not focus on object behavioural tendencies.

Harris [44] proposed using class in addition to allocation site to dynamically predict lifetimes. This work looked at tendencies of classes to become pre-tenured and found that classes which become pre-tenured are closely related in the inheritance hierarchy. Harris stated that it may be possible to identify

a common supertype (possibly an abstract class or an interface), and to share sampling information between all of its subtypes.

2.4.5 Object Connectivity

On the subject of temporal locality, Lam, Wilson, and Moher [45] presented work which groups co-active objects in order to improve temporal locality. A few common data types with common access patterns were targeted for optimisation and the garbage collector was made to recognise data structures headed by commonly used data types in order to adapt the traversal approach. The justification for the assumed access patterns were the author's assumptions and a study of access statistics.

On the subject of co-active objects, optimisations has been achieved from prefetching cache lines in objects pushed onto the mark stack [46]. Hirzel et al. [47] also explored the connectivity of heap objects and discovered that connectivity strongly correlates with object lifetimes and death times. Partitioning objects by connectivity could therefore be advantageous. However the object behaviour analysis in their paper center around lifetime rather than hotness.

Chapter 3

Methods

The motivation behind the design of ThinGC is that, due to locality of reference, objects classified as hot or cold will likely retain their classification. Consequently the effort to classify and process objects according to their hotness will be justified.

This thesis will examine whether this locality of reference can be observed by studying objects as they are relocated to various memory addresses and classified as hot or cold. Properties of the objects such as class and size will also be taken into account to examine whether common trends can be observed for objects exhibiting similar properties.

This chapter consists of a brief overview of the method (section 3.1), a more detailed and technical description (section 3.2), assessment of the reliability of the method (section 3.3), reflection of application to other areas (section 3.4) and estimation of limitations of the method (section 3.5).

3.1 Overview

The collection and processing of object information consists of three steps. In the first step, object information such as class, size, hotness and current address for each object, along with information on all forwarded addresses, is logged in each GC cycle of ThinGC. This process is described in detail in subsection 3.2.1. Information produced by the same object in each GC cycle is then compiled in a second step by following the initial object address as the object is moved by the compacting GC (ThinGC) throughout its lifetime. This process is described in detail in subsection 3.2.2. This compiled object information is then, in the third step, used to create plots and calculate metrics used to analyse object behaviour. This process is described in detail in

subsection 3.2.3.

3.2 Implementation

Any implementation described in the following sections was realised specifically for the purpose of this thesis.

3.2.1 Logging

In order to follow an object through its life and potentially many addresses, at the beginning of the ThinGC cycle, in STW1 when the forwarding table from the previous cycle is complete, the from-to address information of each forwarding in the forwarding table is logged.

The object information such as address, hotness, class and size for objects on live pages is logged in STW3, right before the relocation phase. The forwarding and page table information is logged during STW phases when application threads are not executing in order to provide accurate information for that cycle.

The logging is performed by a method added to ThinGC (`ZHeap::log_hot_cold_data()`), which is located so that it has access to the page table and relocation set.

`ZHeap::log_hot_cold_data()` is called from the class which directs the major phases of ThinGC.

In STW1, `ZHeap::log_hot_cold_data()` processes all forwardings in the relocation set (with a relocation set iterator) and runs each forwarding through another added method (`ZForwarding::log_hot_cold_data()`) which skips empty forwarding entries, gets the to-offset and from-index from the forwarding entry, converts the entry page table/from-index into the from-address and logs the to and from-addresses along with the cycle number using existing logging functionality.

In STW3, `ZHeap::log_hot_cold_data()` processes all pages in the page table. For all objects within live pages which were not allocated after the latest mark phase, the function checks if an object is hot in a "hotmap" using its address, determines the class and size of the object and logs the object address, hotness, size, class and the cycle number using the same logging functionality as in STW1.

Figure 3.1 shows when the forwarding and page tables are logged.

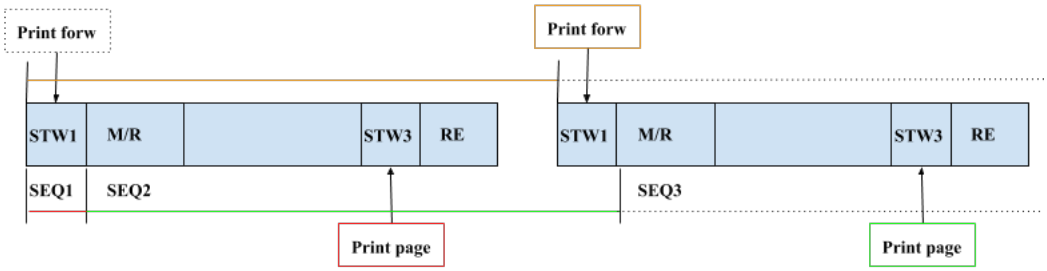

Figure 3.1: Logging from ThinGC.

The size of objects is determined to be “small”, “medium” or “large” by globals in ZGC. The limit of small objects depends on the size of small pages which in turn depend on the `ZPlatformGranuleSizeShift` global which in this case is 21, allowing 2MB for small pages and 2.5KB for small objects. `ZHeap::log_hot_cold_data()` logs the forwarding and page table information into the same file with identifying rows specifying if forwarding or page table information are to follow.
In Table 3.1 the overhead of the logging can be seen in time and number of extra GC-cycles. We see that the number of extra cycles is usually only one.

	# cycles	**# cycles**	**Time real rounded (s)**	**Time real rounded (s)**
Benchmark	*Logging*	*No logging*	*Logging*	*No logging*
Avrora	5	4	5	5
Fop	5	4	6	3
H2	9	9	142	16
Jython	8	7	22	11
Luindex	3	2	3	2
Lusearch	7	6	5	4
Pmd	7	7	9	4
Sunflow	7	7	8	5
Tradebeans	10	8	39	20
Xalan	6	5	7	5

Table 3.1: Overhead of logging in time (s) and number of GC cycles.

3.2.2 Compiling Hotness Information

After the object hotness information has been logged from ThinGC, the forwarding table and page table information are combined. The address and hotness information of each object is collected into a story which spans its lifetime. This is done by a separate script (`process_trace.rb`) which processes the log file into an "object file" with one row of information for each object. The results are saved to file in order to limit executions. This also enables other potential scripts to be run on this compiled data which we call an "object file".
The script grows a persistent hashmap containing one triplet of cycle data [cycle, address, hotness] for every cycle each object has been alive. The triplets of cycle information are associated with the stringified object identifier (cycle, address, size and class information).
When processing objects, the object identifier is associated with the current address of the object in a "current cycle" hashmap. Cycle triplets are also associated with their corresponding object identifiers in the aforementioned persistent hashmap.
When processing a forwarding, the object identifier in the hashmap for the current cycle is re-associated with the "destination" address of the forwarding. When processing the next cycle, the hashmap from the previous cycle containing object identifiers is preserved. In this preserved hashmap, identifiers for forwarded objects are found at their new addresses, the identifiers are transferred to the hashmap for the current cycle and the triplets of cycle data are once again associated with the object identifiers in the persistent hashmap.
In the implementation of this first script, edge cases are considered such as edge-case one: two different objects being forwarded to and from the same address in the same cycle and edge-case two: one object dying at an address in the same cycle as another object is forwarded to that same address.
Table 3.2 shows the time and memory needed to run (`process_trace.rb`) on each of the ten selected benchmarks.

process_trace.rb		
Benchmark	**Time real rounded (s)**	**Mem max rounded (MB)**
Avrora	1	63
Fop	7	455
H2	407	5465
Jython	16	578
Luindex	1	49
Lusearch	2	60
Pmd	10	416
Sunflow	3	165
Tradebeans	41	1091
Xalan	2	65

Table 3.2: Runtime and peak memory usage of the first script (`process_trace.rb`) which compiles hotness information.

3.2.3 Calculating Hotness Metrics

Each row of object lifetime information in this newly created object file is then processed by a second script (`process_obj.rb`) which calculates statistics for the objects and the program.
Calculations in the second script include: the number of hot and cold objects in each ThinGC cycle, the number of transitions (either from hot to cold or from cold to hot) for each object, the number of reheats (recall: transitions specifically from cold to hot) for each object and the longest consecutive streak for each object, expressed as a percentage of the object lifetime in GC cycles. When calculating consecutive hot/coldness, only the longest consecutive streak of an object is considered.
When collecting consecutive streak information for each object in a benchmark, the object is counted as an object which has had a longest consecutive streak lasting a certain number of GC streaks. This streak length is always between 0 and the maximum number of GC cycles of the program. When expressing this streak as a percentage of the object lifetime however, the streak can be anywhere between and including 0% and 100% of object lifetime. The object is therefore counted as an object which has had a longest consecutive

streak lasting within a certain ten-percentage interval of its lifetime. Finally we can then produce information on the number of objects which has had longest hot/cold streaks lasting within a certain interval. The intervals are: (0,10), [10-20), [20-30), [30-40), [40-50), [50-60), [60-70), [70-80), [80-90) [90-100) and 100.
These metrics were chosen to give an overview of object behaviour and consistency. The metrics were computed for the set containing all objects as well as the subsets containing small, medium and large sized objects, and for the subsets containing objects of each one of the classes in the given program.
When all rows of object information have been processed, the most common classes are determined by count and kept for future comparison between programs.
Four types of plots, which will be presented in the results section, are printed presenting the results from the metric calculations. The plots are printed for the set of all objects, for the size subsets (small, medium and large) and for the the subsets of the most common classes. The plots were created using Pyplot [48] and can be used to get an overview of program behaviour, compare behaviour of different subsets of objects within the same program or the same subset of objects across programs.
In order to more efficiently compare statistics across programs, weighted averages for the given program are also calculated. Metrics include: average longest consecutive streak (as a percentage of program lifetime in GC cycles), average number of reheats (as a percentage of the maximum possible number of reheats the object could have made), and average percentage of objects with a "long" longest consecutive streak. A consecutive streak of an object was defined as "long" if it exceeded 50% of the object lifetime.
These averages were calculated for the set containing all objects, for the subsets containing small, medium and large sized objects and for the subsets containing objects of the most common classes in the program.
These average metrics were chosen in order to investigate whether locality of reference could be observed. If indeed objects in cold storage are likely to remain cold, a larger percentage of objects will be consecutively cold for what could be considered a long time (more than 50% of the object's lifetime in this case) and transitions from cold to hot would not be common.
Table 3.3 shows the time and memory needed to run (`process_obj.rb`) on each of the ten selected benchmarks.

process_obj.rb		
Benchmark	**Time real rounded (s)**	**Mem max rounded (MB)**
Avrora	31	339
Fop	42	340
H2	218	341
Jython	49	347
Luindex	30	337
Lusearch	32	339
Pmd	43	340
Sunflow	34	339
Tradebeans	67	355
Xalan	31	339

Table 3.3: Runtime and memory usage of the second script (`process_obj.rb`) which calculates metrics and produces plots and tables.

3.3 Reliability

The reliability of the logging from ThinGC is tied to the reliability of existing code developed and tested by Oracle. The added code mimics already existing examples of how to correctly iterate forwarding and page tables.
The reliability of the first script was ensured by 4 tests. A test to determine that, in the summation of an object's lifetime, the difference between two consecutive cycle numbers is always one. A test to validate each forwarding in the completed summation of an object lifetime. A test to confirm that all objects from the HCG-log are summarised, and a test to assure that there are no duplicates of the unique object identifier (object address together with the cycle in which the object was created). The first script was also implemented in two languages with two different approaches to see that both implementations would produce the same result.

3.4 Application to Other Areas

Since mutator accesses classify objects as hot or cold, the results are independent from the specific garbage collector used and can be applied in any situation where object behaviour information is applicable.
The results do rely on the ZGC size globals to classify objects as small, medium or large. This can easily be modified however, to introduce custom size definitions when collecting data from an another source. Alternatively, the pure object size can be collected when logging and a classification of size can be added in one of the current scripts.

3.5 Limitations

Logging the information from ThinGC slightly increases the total number of GC cycles for most programs we collected information from. A report of the number of GC cycles with and without logging is included in Table 3.1. This could have an effect on the metric calculations, however we deemed the increase in cycles to be slight enough not to effect measurements significantly.
Since the first script collects object information from the log file into a hashmap which is kept in memory and printed to file only after an entire log file is processed, processing the log files with the first script can require a significant amount of memory when handling a program with a large number of objects, see Table 3.2.
The first script also expects the log file from ThinGC to have a certain layout. If using data collected any other way, either the first script or the alternatively collected data will need to be modified.
Since the second script processes the compiled information of each object lifetime, the runtime of the second script can be significant when handling object files for programs with a large number of objects, see Table 3.3.
The second script is adapted for the needs of this thesis, plots and tables are not produced from user input. Editing of the second script will be required in order to produce additional plots and tables.
The metrics calculated by the second script are initial ideas and can serve as a starting point. Modifications can be done, depending on what purpose the data is being collected for. For example experimenting with what classifies as a long consecutive streak, changing the classification of ”consecutive” to allow relatively short interruptions, or collecting information about the second longest consecutive streak.

More specific research questions could be explored by collecting more focused data. The behaviour of a specific class could be explored. Since the second script produces results for the five most popular classes from each program, the collection is not guaranteed to contain statistics for a specific class in a program. The collection could be modified to examine classes of interest, specified by the user.
This paper contains only data collected from the DaCapo benchmark suite [49] which forms a small sample size. The method of extending this sample is however presented in this thesis.

Chapter 4

Results

In this chapter we present the plots and tables produced as a result of the study. The chapter consists of section 4.1: a brief summary of the results to come, section 4.2: a presentation of the benchmark-suite used and the objects it contains, section 4.3: an in depth description and presentation of plots, and section 4.4: an in depth description and presentation of tables.
The values in the tables are rounded to two decimal places to improve readability.

4.1 Overview

Four types of plots were created to present the results. First, The "Reheats" plot, which shows the eagerness of objects to transition from cold to hot. Second, the "Ratios" plot, where the percentage of hot and cold objects in each GC cycle is displayed. Third and fourth, the "Streaks" and "Main temperature" plots, which shows two ways of expressing the lengths of each object's longest hot and cold streak.
From the "Reheats" plots we can determine if objects have a tendency to remain cold once they have become cold. If objects do generally remain cold, the number of reheats should be low for a high percentage of objects.
From the "Ratios" plots we can, for example, observe the steadiness of cold objects, in what cycles (if any) the number of hot or cold objects increase or decrease, and the ratio of hot and cold objects in a given cycle.
From the "Streaks" and "Main temperature" plots we can deduce if long cold streaks are more common than long hot streaks and how long these longest hot and cold streaks tend to be, both as a number of cycles and as an approximate percentage of object lifetime.

Three tables of average metrics were also composed in order to compare benchmarks. The following metrics populate the tables: average number of reheats, the average length of the longest consecutive hot/cold streak, and the average percentage of ”mostly” hot/cold objects in a program.
In these plots and tables, object and program lifetimes are always expressed as a number of GC cycles.
The plots and averages were produced and calculated separately for objects of small, medium and large size (as classified by ZGC), for objects of the five most common classes in each program and for all objects combined.

4.2 Benchmark suite

Benchmarking was done on an Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz with 4 cores (2 hyper-threads/core), 32KB L1, 256KB L2, 3072KB L3, 5.7 GB RAM, Running Ubuntu 16.04 LTS with Linux kernel version 4.15.0-74-generic (x86_64) and the C/C++ compiler used was GCC 7.4.0.
The chosen benchmark suite is the DaCapo benchmark suite [49] version 9.12-MR1-batch. The chosen benchmarks from the suite, along with heap sizes and number of GC cycles of the logged run are presented in Table 4.1.

Benchmark	**Heap size**	**Nr GC cycles**
Avrora	70 Mb	5
Fop	180 Mb	5
H2	1400 Mb	9
Jython	1000 Mb	8
Luindex	120 Mb	3
Lusearch	600 Mb	7
Pmd	300 Mb	7
Sunflow	800 Mb	7
Tradebeans	2000 Mb	10
Xalan	600 Mb	6

Table 4.1: Heap sizes and number of GC cycles of chosen benchmarks.

A study of the types of objects contained in the benchmark suite was made, and concluded that the suite contains mostly small and medium sized objects

by ZGC standards, as can be seen in Table 4.2. Due to this, comparison of objects of different size was hindered.

Benchmark	**Nr. objects**	**Small obj**	**Med obj**	**Large obj**
Avrora	59486	100%	0%	0%
Fop	624203	99,9997%	0,0003%	0%
H2	6093832	99,9997%	0,0002%	0,0001%
Jython	693350	99,9991%	0,0009%	0%
Luindex	66683	100%	0%	0%
Lusearch	102684	100%	0%	0%
Pmd	591435	100%	0%	0%
Sunflow	219098	99,9991%	0,0009%	0%
Tradebeans	1061769	99,9997%	0,0003%	0%
Xalan	60142	100%	0%	0%

Table 4.2: Ratios of objects of different sizes in DaCapo benchmarks.

The prevalence of the top five most popular classes between the benchmarks was also looked into, see Table 4.3. The top 5 most common classes were:

- ”[B”: ByteArray,
- ”String”: java.lang.String,
- ”HashMap$Node”: java.util.HashMap$Node,
- ”Class”: java.lang.Class,
- ”ConcurrentHashMap$Node”: java.util.concurrent.ConcurrentHashMap$Node.

Benchmark	[B	String	HashMap$ Node	Concurrent HashMap$ Node	Class
Avrora	16,24%	15,85%	8,93%	2,64%	2,70%
Fop	7,71%	7,58%	4,36%	0,66%	0,46%
H2	7,16%	7,15%	0,22%	0,07%	0,03%
Jython	18,02%	17,87%	0,84%	8,72%	0,58%
Luindex	15,17%	14,31%	7,56%	5,14%	2,22%
Lusearch	22,73%	21,95%	3,29%	1,77%	1,28%
Pmd	15,12%	15,02%	1,51%	0,43%	0,39%
Sunflow	4,74%	4,60%	2,15%	1,15%	0,90%
Tradebeans	10,75%	10,40%	7,43%	3,81%	1,12%
Xalan	18,21%	17,81%	11,19%	3,06%	3,12%
Average	13,59%	13,25%	4,75%	2,74%	1,28%

Table 4.3: Ratios of objects of different classes in DaCapo benchmarks.

The percentages of objects which were consistently either hot or cold and never transitioned were also collected, see Table 4.4. From this table we can see that, on average in one of the chosen benchmarks, more objects are consistently hot than consistently cold. The percentages for consistently cold objects are more dependable, with a 4,36% difference between the benchmark with the highest (tradebeans, 4,36%) and lowest (sunflow, 0%) value. This suggests that we can be confident that the percentage of consistently cold objects will be close to its average (1%) in the benchmarks chosen. Accordingly most objects are hot during at least one GC cycle.
The percentages of consistently hot objects however seem to be undependable. We see about a 71% difference between the the benchmark with the lowest (jython, 10,49%) and highest (sunflow, 81,45%) value. This highlights a possible factor for suitabilty to the ThinGC approach. We see that in some benchmarks, for example sunflow, the percentage of objects which are ever cold are considerably lower than in other benchmarks, for example in jython.

Benchmark	Hot	Cold
Avrora	27,81%	0,12%
Fop	77,91%	0,60%
H2	18,33%	2,07%
Jython	10,49%	0,25%
Luindex	72,41%	0,04%
Lusearch	61,39%	1,42%
Pmd	56,82%	1,75%
Sunflow	81,45%	0%
Tradebeans	19,68%	4,36%
Xalan	38,42%	0,08%
Average	46,47%	1,07%

Table 4.4: Percentages of consistently hot and cold objects in DaCapo benchmarks.

Continuing the review of the benchmark suite, the average lifetime of objects of different types within the benchmarks were calculated. The lifetimes are expressed as a percentage of the total number of GC cycles in the program. Average lifetimes of objects of different sizes can be found in Table 4.5. Medium sized objects seem to have longer lifetimes on average than small sized objects, medium sized objects are however considerably underrepresented and the measurement could be unreliable.

Benchmark	All	Small	Medium
Avrora	67,54%	67,54%	
Fop	31,72%	31,72%	100%
H2	60,46%	60,46%	44,44%
Jython	43,97%	43,97%	52,08%
Luindex	55,34%	55,34%	
Lusearch	39,38%	39,38%	
Pmd	34,54%	34,54%	
Sunflow	33,93%	33,93%	100%
Tradebeans	60,18%	60,18%	26,67%
Xalan	80,70%	80,70%	
Average	50,78%	50,78%	64,64%

Table 4.5: Average lifetimes of objects (as a percentage of program lifetime).

Average lifetimes of objects of different classes can be found in Table 4.6.

Benchmark	[B	String	HashMap $Node	ConcurrentHashMap $Node	Class
Avrora	68,12%	68,22%	64,07%	90,80%	84,05%
Fop	41,41%	41,22%	36,97%	59,49%	78,17%
H2	58,92%	58,87%	50,16%	98,24%	95,90%
Jython	45,29%	45,01%	84,09%	43,77%	95,51%
Luindex	64,57%	65,68%	66,85%	61,54%	80,65%
Lusearch	39,56%	39,93%	82,70%	97,34%	96,85%
Pmd	31,27%	31,21%	59,96%	52,90%	87,63%
Sunflow	96,34%	96,50%	98,08%	93,31%	97,58%
Tradebeans	74,62%	74,55%	78,40%	70,86%	82,63%
Xalan	90,76%	91,27%	55,24%	95,34%	98%
Average	61,09%	61,25%	67,65%	76,36%	89,70%

Table 4.6: Average lifetimes of objects of different classes (as a percentage of program lifetime).

The average lifetimes of consistently hot and cold objects can be found in Table 4.7 where we see that consistently cold objects have longer lifetimes on average than consistently hot objects.

Benchmark	**Hot**	**Cold**
Avrora	49,07%	79,73%
Fop	24,57%	32,11%
H2	12,11%	49,25%
Jython	38,91%	56,34%
Luindex	38,55%	78,67%
Lusearch	21,58%	14,76%
Pmd	21,94%	35,98%
Sunflow	19,78%	85,71%
Tradebeans	35,21%	39,36%
Xalan	57,00%	60,13%
Average	31,87%	53,20%

Table 4.7: Average lifetimes of consistently hot and cold objects (as a percentage of program lifetime).

4.3 Plots

The plots are constructed to give an overview of objects and provide a way to observe trends in the benchmarks.

4.3.1 Reheats

As previously stated, a reheat is a transition from cold to hot storage. A high percentage of objects reheating few times signifies an overall reluctance of objects to reheat within the benchmark.
In Figure 4.1 (“avrora”), Figure 4.2 (“lusearch”), Figure 4.3 (“sunflow”), Figure 4.4 (“fop”), Figure 4.5 (“luindex”), Figure 4.6 (“jython”) and Figure 4.7 (“xalan”), close to 100% of objects were never reheated and most of the very few remaining objects reheated only once or twice.

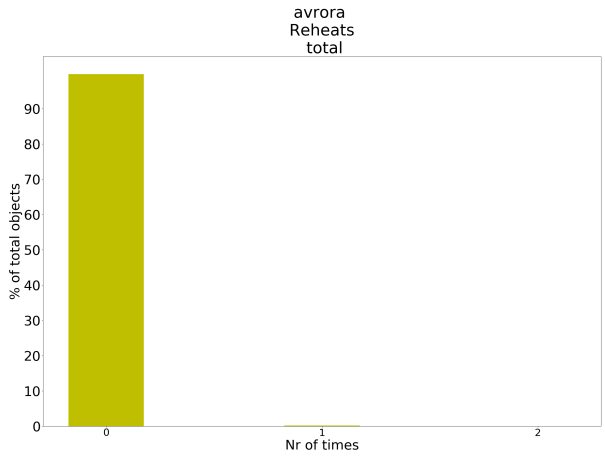

Figure 4.1: Number of reheats
for all objects in the ”avrora” benchmark.

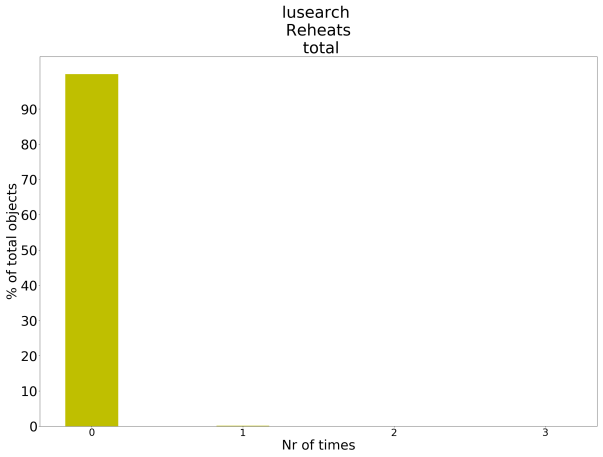

Figure 4.2: Number of reheats
for all objects in the ”lusearch” benchmark.

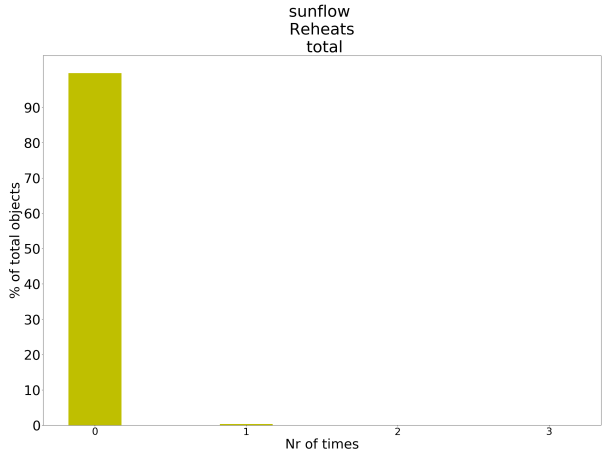

Figure 4.3: Number of reheats
for all objects in the ”sunflow” benchmark.

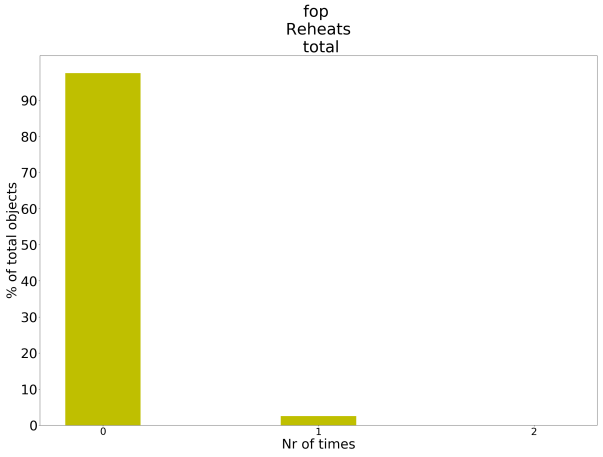

Figure 4.4: Number of reheats
for all objects in the ”fop” benchmark.

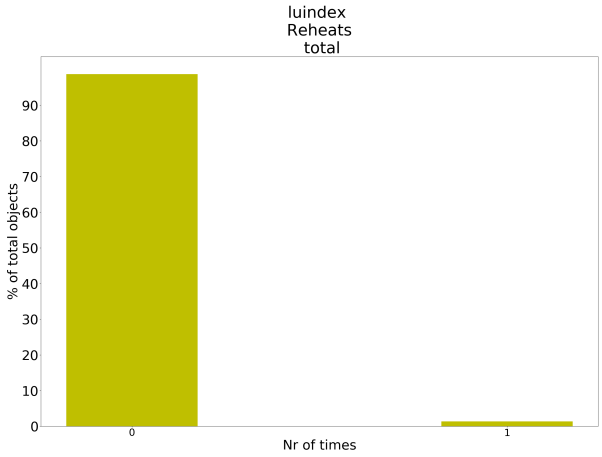

Figure 4.5: Number of reheats
for all objects in the ”luindex” benchmark.

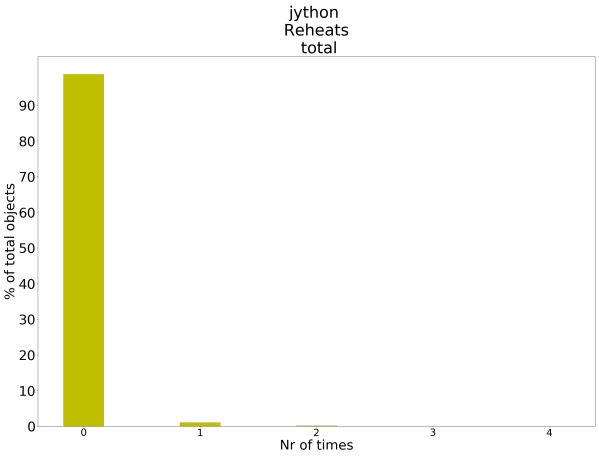

Figure 4.6: Number of reheats
for all objects in the ”jython” benchmark.

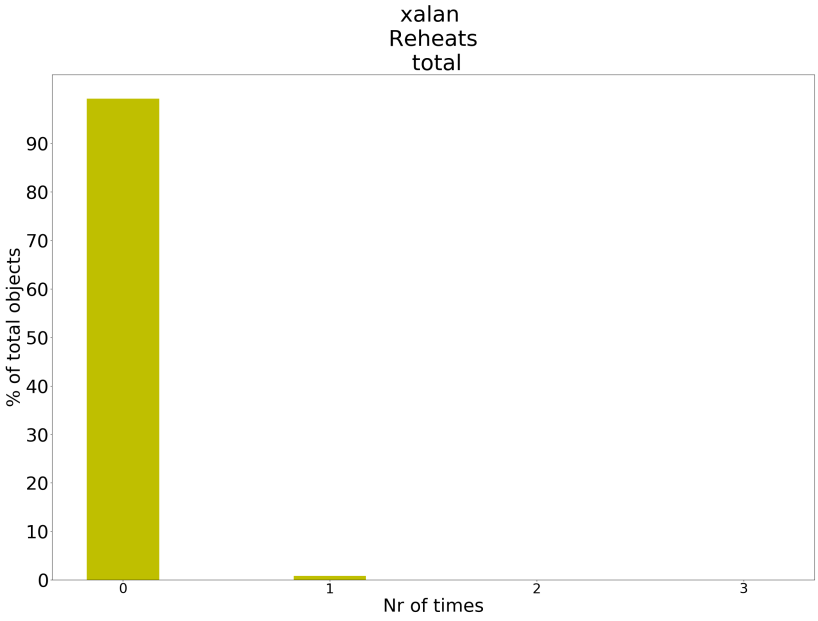

Figure 4.7: Number of reheats
for all objects in the ”xalan” benchmark.

In Figure 4.8 (“tradebeans”), Figure 4.9 (“pmd”) and Figure 4.10 (“h2”), the percentages of objects reheating more than once were higher than in figure 4.1-4.7, but for Figure 4.8 (“tradebeans”) and Figure 4.9 (“pmd”) the

number of reheats is still relatively low. In Figure 4.10 (“h2”) however, we see that a significant percentage of objects (about 42%) reheat twice which is a considerable number of reheats given the program lifetime of “h2” in GC cycles.

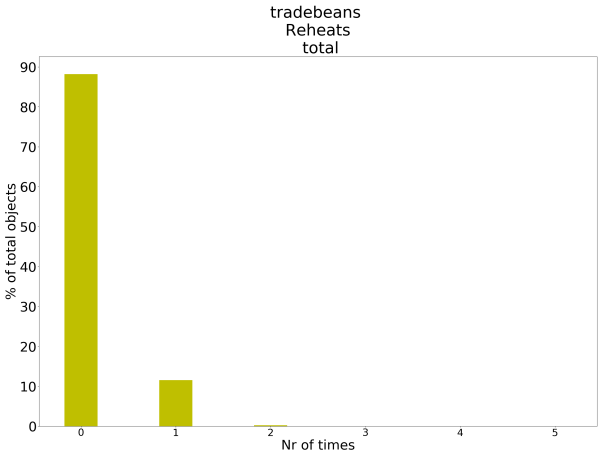

Figure 4.8: Number of reheats
for all objects in the ”tradebeans” benchmark.

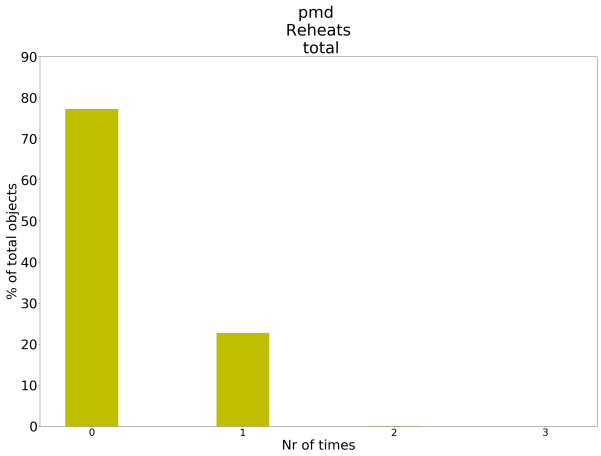

Figure 4.9: Number of reheats
for all objects in the ”pmd” benchmark.

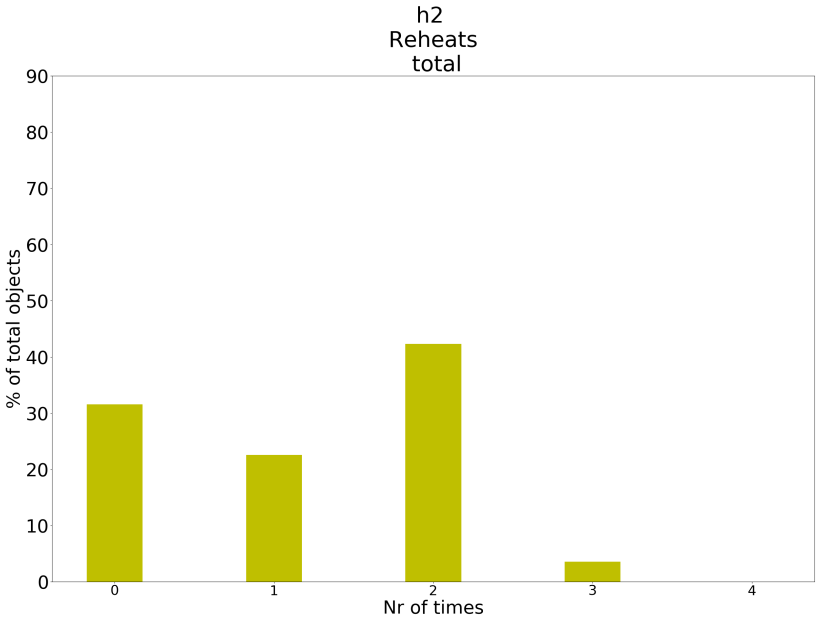

Figure 4.10: Number of reheats
for all objects in the ”h2” benchmark.

These plots suggest that the risk of an object reheating once it has become cold is low and that objects tend to stay cold once they have become cold.

4.3.2 Ratios

When plotting all objects across all GC cycles of a benchmark, we can observe general trends for hot and cold objects throughout the benchmark runtime. Viewing the data in this form however, we don't know for certain what percentage of the cold objects stay cold. Even if the bars from one cycle to the next are identical, two equally sized groups of objects of opposite hotness could both have transitioned in the time between cycles. Thanks to the "Reheats" plots however, we know that the probability of objects reheating is low, and plotted cold bars in adjacent GC cycles are probably referring to the same objects.
For most benchmarks, especially Figure 4.11 ("tradebeans"), Figure 4.12 ("xalan"), Figure 4.13 ("lusearch") and Figure 4.14 ("sunflow"), cold objects usually seem to stay steadily cold, neither decreasing nor increasing drastically in numbers once they have become cold. For these benchmarks, cold objects also seem to represent a significant percentage of the objects. Exceptions

where the percentage of cold objects is not as significant are Figure 4.15 (“fop”) and Figure 4.16 (“luindex”), nevertheless the cold objects in these benchmarks still seem to stay steadily cold.

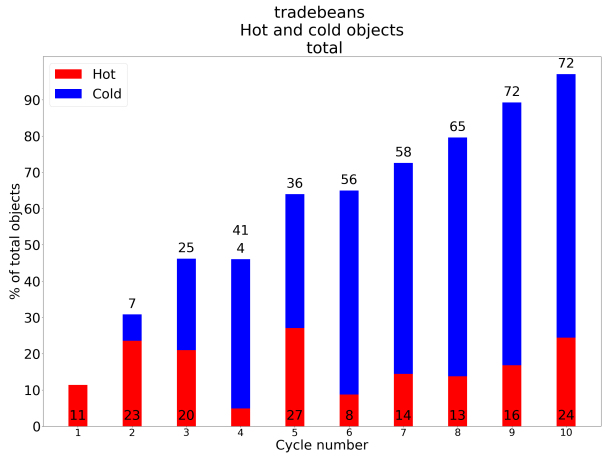

Figure 4.11: Number of hot and cold objects in each GC cycle of the ”tradebeans” benchmark.

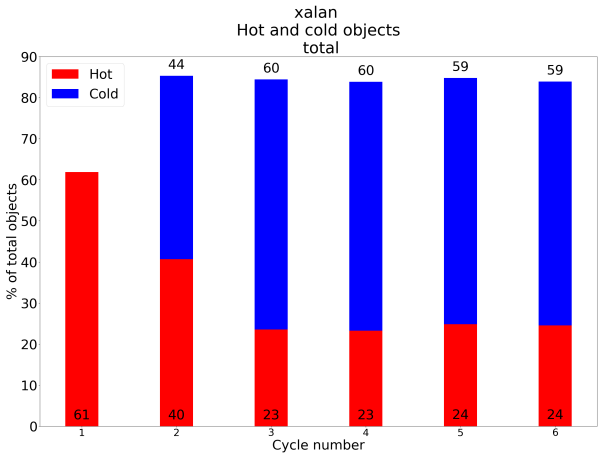

Figure 4.12: Number of hot and cold objects in each GC cycle of the ”xalan” benchmark.

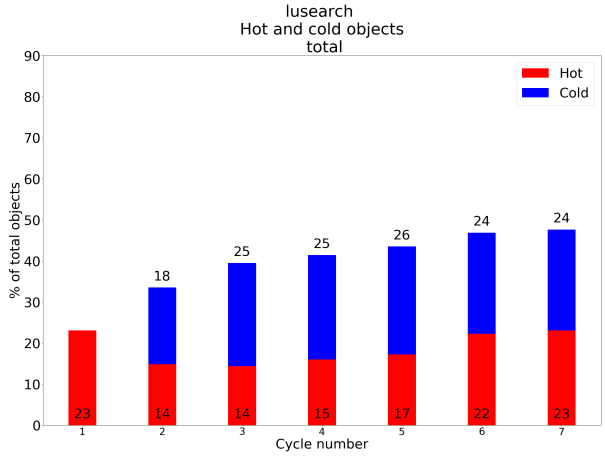

Figure 4.13: Number of hot and cold objects in each GC cycle of the ”lusearch” benchmark.

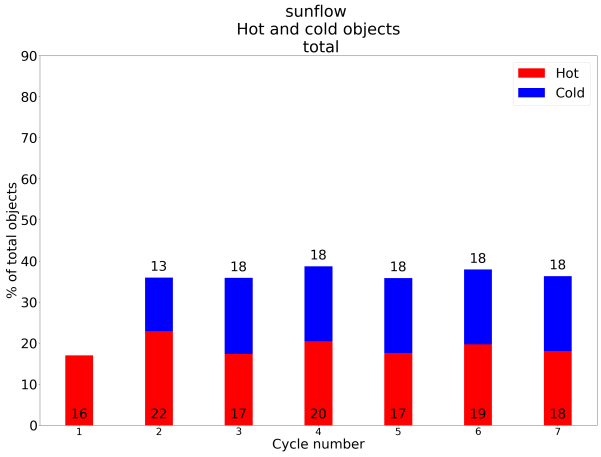

Figure 4.14: Number of hot and cold objects in each GC cycle of the ”sunflow” benchmark.

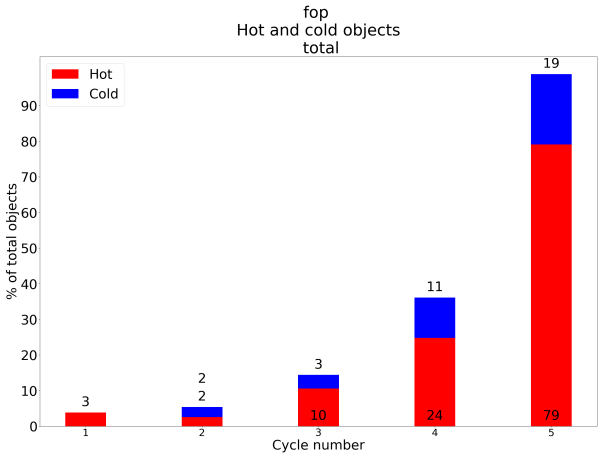

Figure 4.15: Number of hot and cold objects in each GC cycle of the ”fop” benchmark.

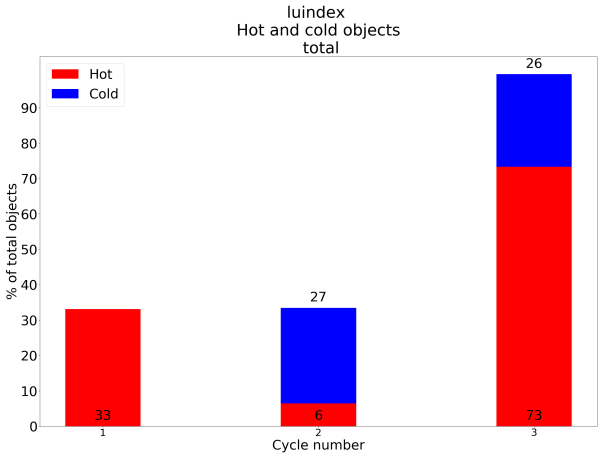

Figure 4.16: Number of hot and cold objects in each GC cycle of the ”luindex” benchmark.

We find in Figure 4.17 (“avrora”) and Figure 4.18 (“jython”) that cold objects do not only steadily stay cold like previously mentioned for figures 4.11-4.16, but the number of cold objects also increase towards the end of the program runtime.

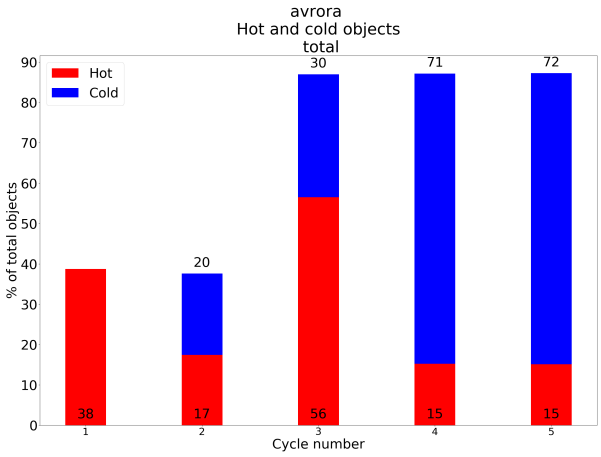

Figure 4.17: Number of hot and cold objects in each GC cycle of the ”avrora” benchmark.

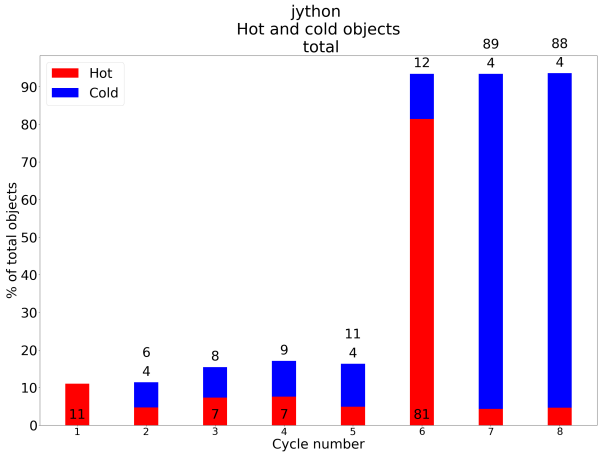

Figure 4.18: Number of hot and cold objects in each GC cycle of the ”jython” benchmark.

Another interesting pattern of behaviour can be found in Figure 4.19 (“pmd”) and Figure 4.20 (“h2”) where we still see cold objects staying steadily cold

with an increase of cold objects towards the end of the program lifetime. In these plots however we also see objects possibly reheating in the very last cycle.

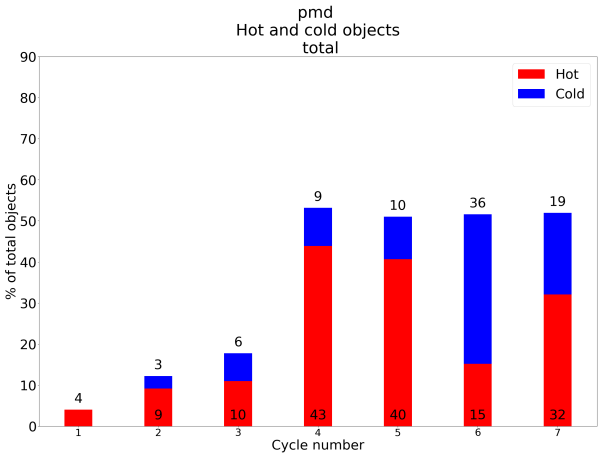

Figure 4.19: Number of hot and cold objects in each GC cycle of the ”pmd” benchmark.

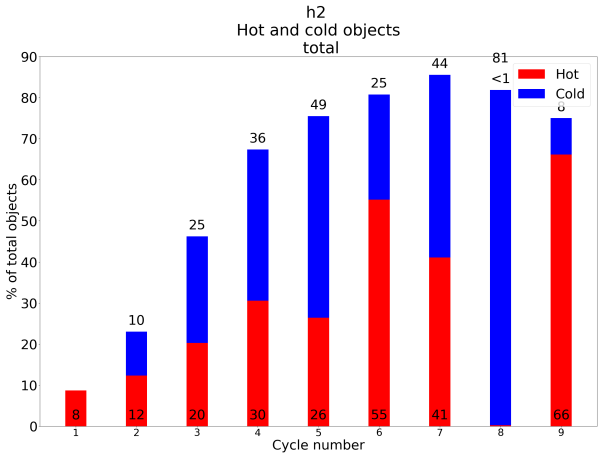

Figure 4.20: Number of hot and cold objects in each GC cycle of the ”h2” benchmark.

4.3.3 Streaks

These plots show the lengths of hot and cold streaks of objects in a program. A streak is defined as a number of GC cycles where the object stayed hot or cold. Only the longest hot and cold streak of each object is considered.
In most plots of the DaCapo benchmarks below, (Figure 4.21 (“jython”), Figure 4.22 (“avrora”), Figure 4.23 (“sunflow”), Figure 4.24 (“xalan”), Figure 4.25 (“tradebeans”), Figure 4.26 (“lusearch”) and Figure 4.28 (“h2”)) we can see that long cold streaks are more common than long hot streaks. In fact objects’ longest hot streaks are most often only one GC cycle long. Any significant percentage of longest cold streaks of length one can only be found in Figure 4.25 (“tradebeans”), Figure 4.26 (“lusearch”) and Figure 4.28 (“h2”) where they are still far outnumbered by length one hot streaks.

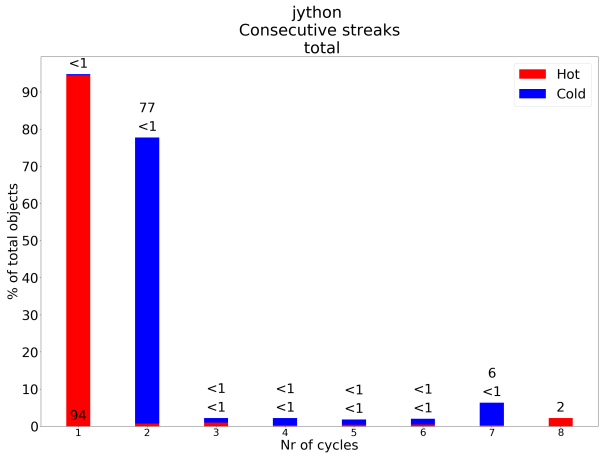

Figure 4.21: Consecutive hot and cold streak lengths of objects in the ”jython” benchmark.

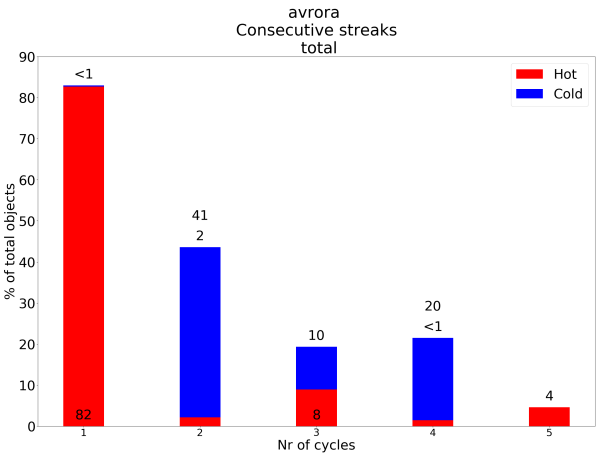

Figure 4.22: Consecutive hot and cold streak lengths of objects in the ”avrora” benchmark.

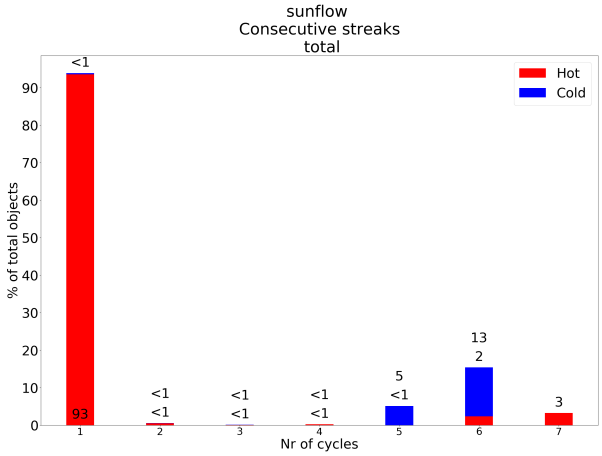

Figure 4.23: Consecutive hot and cold streak lengths of objects in the ”sunflow” benchmark.

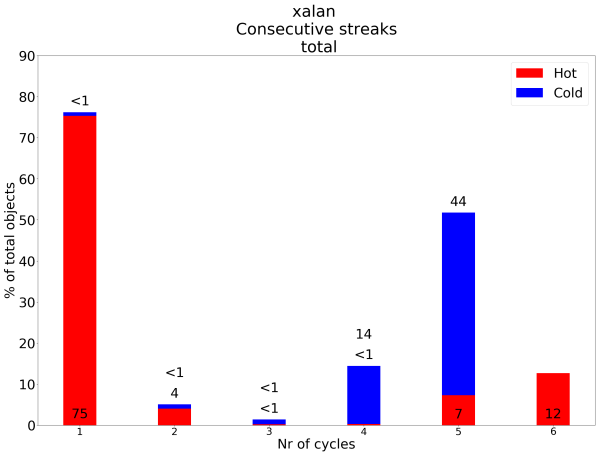

Figure 4.24: Consecutive hot and cold streak lengths of objects in the ”xalan” benchmark.

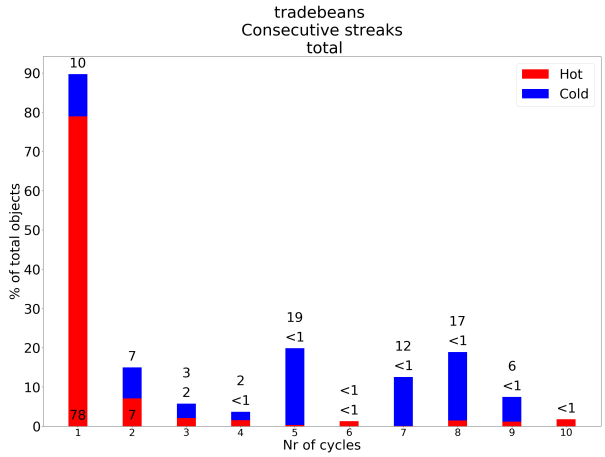

Figure 4.25: Consecutive hot and cold streak lengths of objects in the ”tradebeans” benchmark.

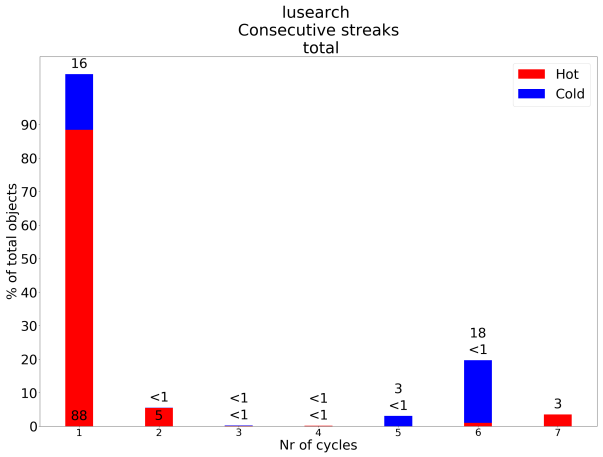

Figure 4.26: Consecutive hot and cold streak lengths of objects in the ”lusearch” benchmark.

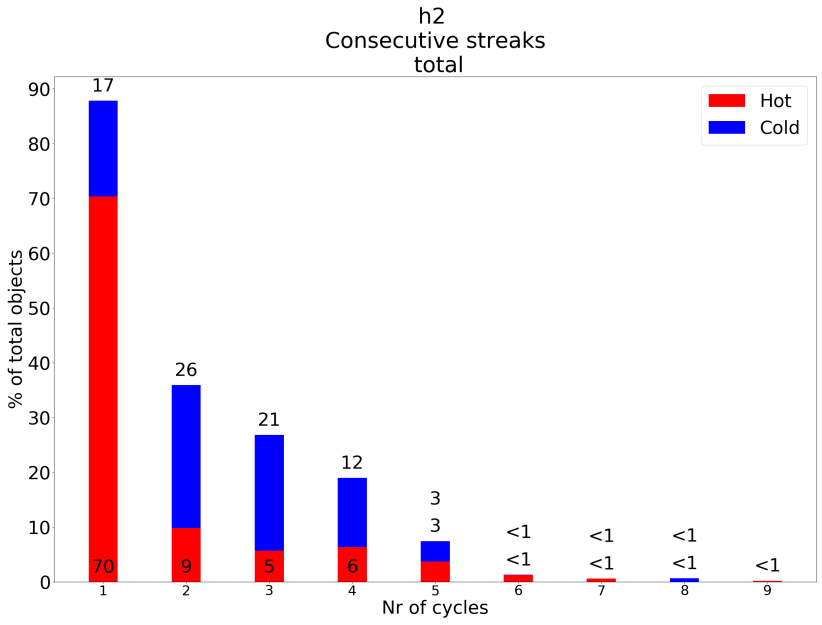

Figure 4.27: Consecutive hot and cold streak lengths of objects in the ”fop” benchmark.

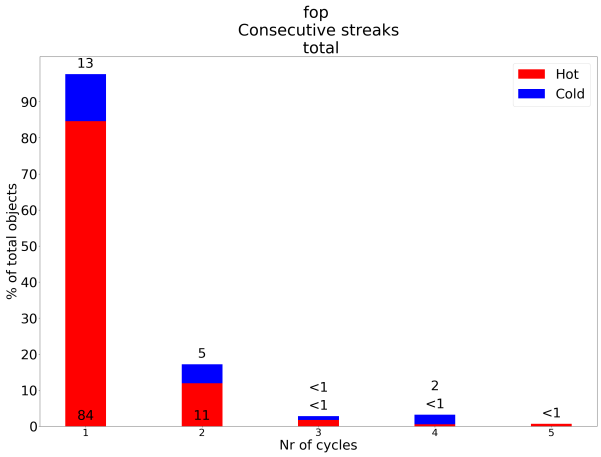

Figure 4.28: Consecutive hot and cold streak lengths of objects in the ”h2” benchmark.

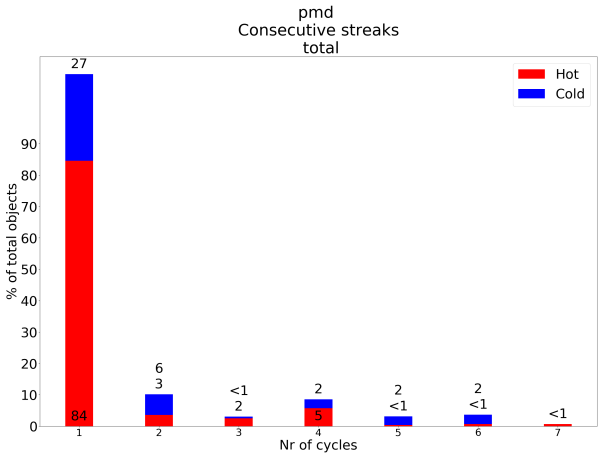

Figure 4.29: Consecutive hot and cold streak lengths of objects in the ”pmd” benchmark.

From these plots we also see that streaks lasting an entire program lifetime are very rare and are only ever hot streaks.
These plots however do not take into account and compare the streak to the object lifetime and so the following plot is also presented:

4.3.4 Main Temperature

In these plots, the lengths of the longest hot and cold streaks of each object are expressed as a percentage of the object lifetime.
When taking into account the object lifetime, the plots show that it is common for objects to have hot streaks which lasts their entire lifetime. This is consistent with the information presented in Table 4.4. Something to consider along with this result is that the examination of the DaCapo benchmark showed the average lifetime of consistently hot objects to be fairly short (27% of program lifetime), as can be seen in Table 4.4.

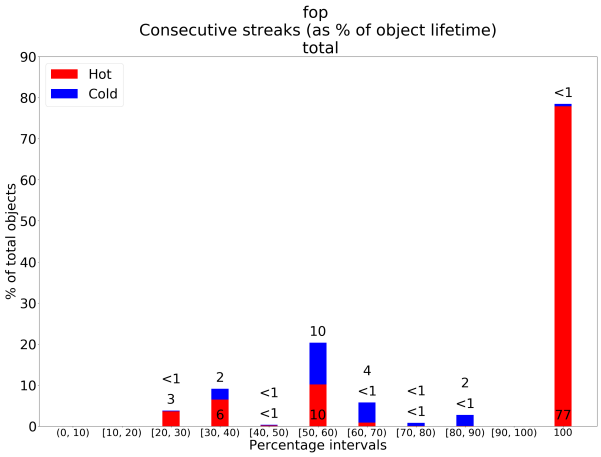

Figure 4.30: Consecutive hot and cold streak lengths of objects in the ”fop” benchmark (as a percentage of object lifetime).

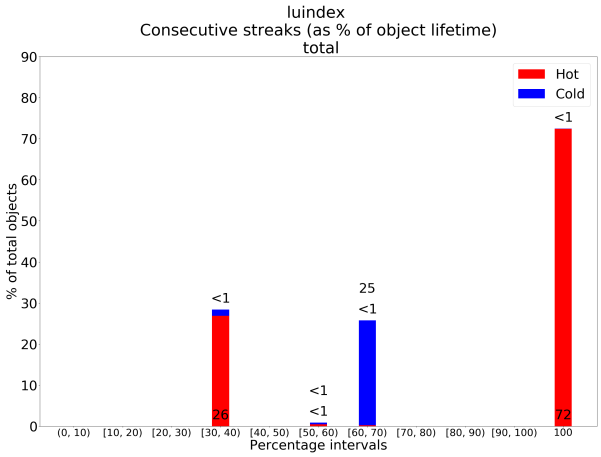

Figure 4.31: Consecutive hot and cold streak lengths of objects in the ”luindex” benchmark (as a percentage of object lifetime).

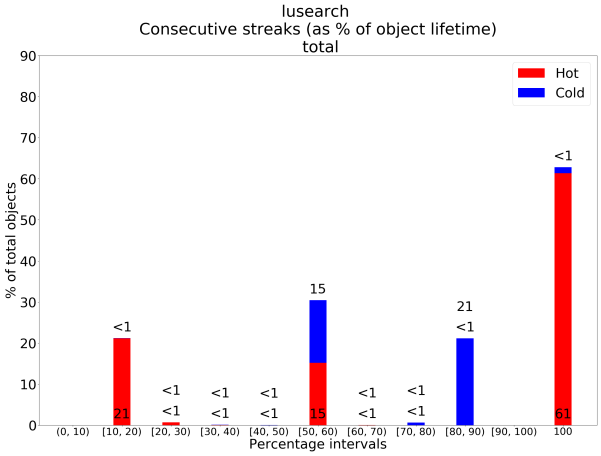

Figure 4.32: Consecutive hot and cold streak lengths of objects in the ”lusearch” benchmark (as a percentage of object lifetime).

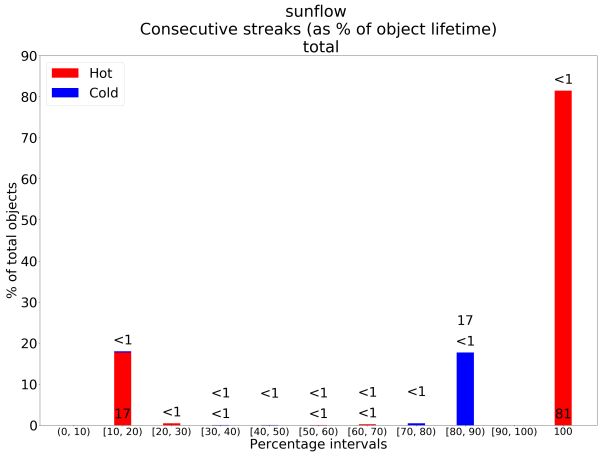

Figure 4.33: Consecutive hot and cold streak lengths of objects in the ”sunflow” benchmark (as a percentage of object lifetime).

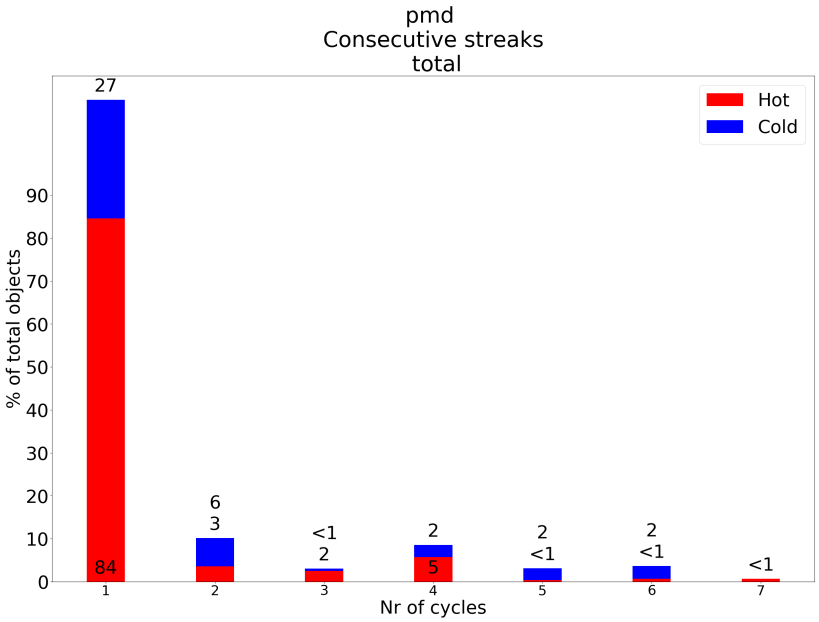

Figure 4.34: Consecutive hot and cold streak lengths of objects in the ”pmd” benchmark (as a percentage of object lifetime).

Most objects which are not consistently hot have longest hot streaks lasting no longer than 60% of object lifetime. Most longest cold streaks however lasts longer than 60% of object lifetime as we can see in Figure 4.35 (“avrora”), Figure 4.36 (“jython”), Figure 4.37 (“tradebeans”) and Figure 4.38 (“xalan”).

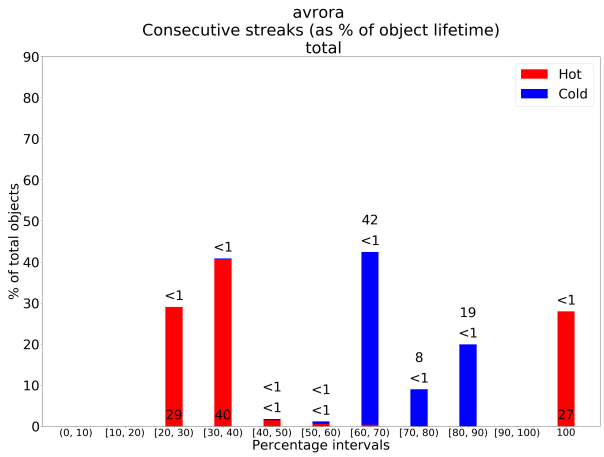

Figure 4.35: Consecutive hot and cold streak lengths of objects in the ”avrora” benchmark (as a percentage of object lifetime).

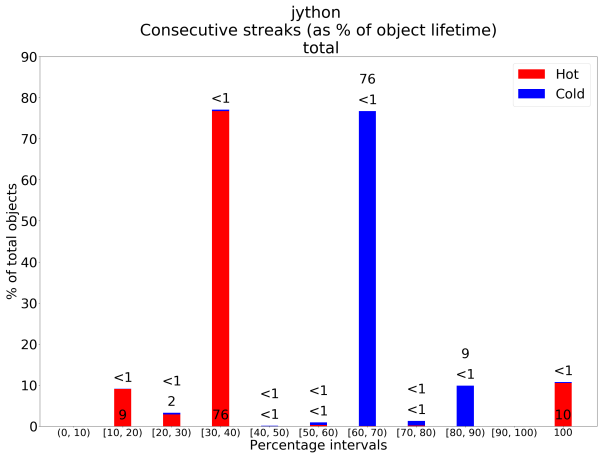

Figure 4.36: Consecutive hot and cold streak lengths of objects in the ”jython” benchmark (as a percentage of object lifetime).

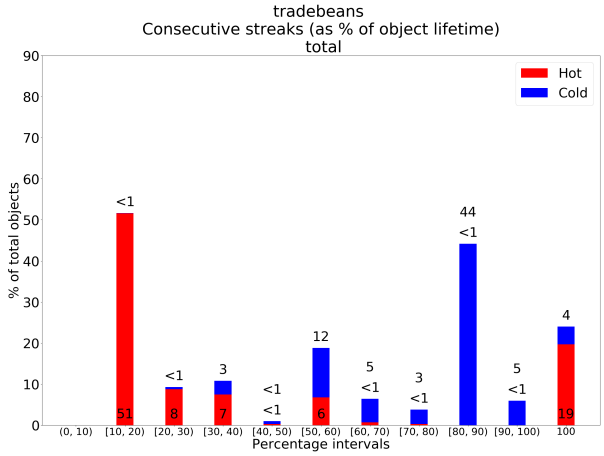

Figure 4.37: Consecutive hot and cold streak lengths of objects in the ”tradebeans” benchmark (as a percentage of object lifetime).

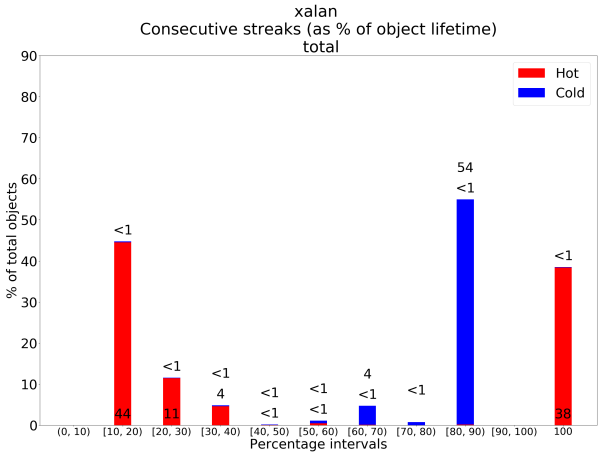

Figure 4.38: Consecutive hot and cold streak lengths of objects in the ”xalan” benchmark (as a percentage of object lifetime).

The one and only exception where most longest cold streaks are not longer than 60% of object lifetime is Figure 4.39 (“h2”) below:

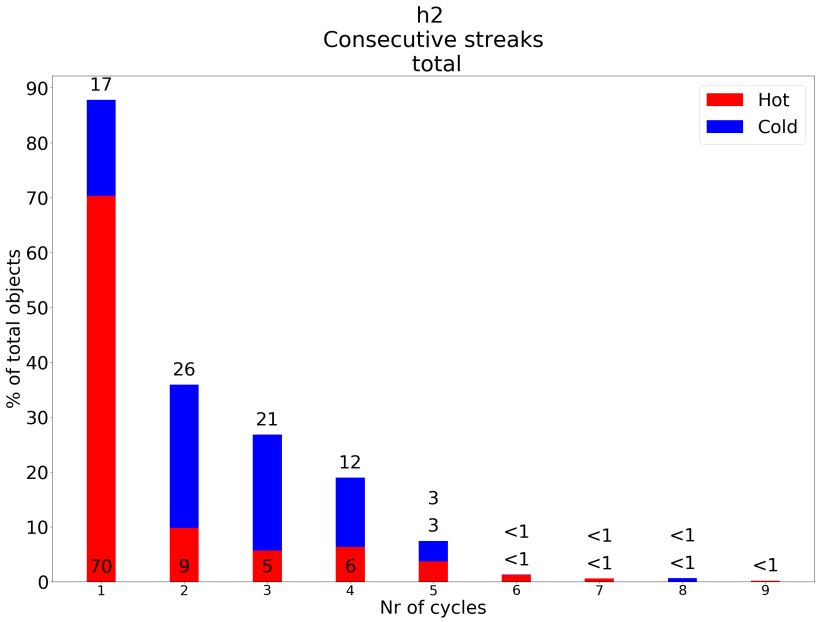

Figure 4.39: Consecutive hot and cold streak lengths of objects in the ”h2” benchmark (as a percentage of object lifetime).

As we have seen it is common for objects to have hot streaks lasting only 30% to 40% of its lifetime and also common for the longest hot streak of an object to last an entire object lifetime. Hot streaks seem to usually either last entire lifetimes, or just be pre/interludes to cold streaks. Our knowledge from the previous plots, along with the fact that most objects which are consistently hot are short lived leads us to assume that most hot streaks are short. We also see that long cold streaks in relation to object lifetime are common (as in Figure 4.32 (“lusearch”), Figure 4.33 (“sunflow”), Figure 4.36 (“jython”), Figure 4.37 (“tradebeans”) and Figure 4.38 (“xalan”)).

4.4 Tables

The tables of averages were created to facilitate comparison between, and observe trends across benchmarks. The trends observed with cross-benchmark averages are streak length, expressed as a percentage of program or object lifetime, either for all objects combined or for specific classes, and reheat tendency. The averages were calculated to give an insight into object stability in terms of hotness across benchmarks.

The percentages of consistently hot or cold objects, originally from Table 4.4, is included in the ”Consistently hot” column to provide context to the average streak lengths.
The chosen benchmarks of the DaCapo benchmark suite contains very few objects which ZGC would classify as large, so sufficient measurements could not be collected to include large object behaviour in the following tables. Since most objects are small, the values for small sized objects and all objects are very similar and only the values for all objects are included in the tables.
Chosen DaCapo benchmarks which do not contain medium sized objects have also been excluded from the average calculation in order to give a more accurate depiction of how medium sized objects behave in programs where they do appear. Most chosen DaCapo benchmarks however contain very few medium sized objects as can be seen in Table 4.2.
Classes were chosen from among the top ten most common classes for each benchmark. The aim was to get as many measurements as possible for each chosen class and present classes that were present in as many of the chosen benchmarks as possible. As a result of this approach, all classes are present in all chosen benchmarks.
In the ”diff” row, the average for a specific size or class in a benchmark is compared to the average for all objects combined, and displayed as the difference between the two averages.
The classes included in the tables are:

- ”[B”: ByteArray
- ”String”: java.lang.String
- ”HashMap$Node”: java.util.HashMap$Node
- ”Class”: java.lang.Class
- ”ConcurrentHashMap$Node”: java.util.concurrent.ConcurrentHashMap$Node

4.4.1 Average Consecutive Streak

The “average consecutive streak” metric expresses the weighted mean of the longest consecutive streak of all objects in each program, expressed as a percentage of the program lifetime in GC cycles. Longest consecutive streaks of length 0 have been excluded from the average calculation in order to give a better understanding of the average length of longest hot/cold streaks when they do occur. A long average longest consecutive hot/cold streak would then

give an assurance of object hot/cold stability throughout the program runtime. Being aware of an average object hot/cold stability factor could contribute to predicting reheats and avoiding wasteful freezes.

Hot Streaks				
Benchmark	**All objects**	**Consistently hot**	**Medium sized objects**	**[B**
avrora	28,59%	27,81%		20,90%
fop	23,92%	77,91%	50%	21,43%
h2	18,66%	18,33%	20,51%	11,34%
jython	15,57%	10,49%	22,91%	12,85%
luindex	37,19%	72,41%		33,90%
lusearch	18,95%	61,39%		15,41%
pmd	19,42%	56,82%		15,71%
sunflow	18,96%	81,45%	57,14%	14,86%
tradebeans	16,16%	19,68%	26,66%	11,85%
xalan	33,01%	38,42%		22,61%
average	23,04%	46,47%	35,44%	18,09%
diff	$\pm$ 0	-	+12,40%	-4,96%
Benchmark	**String**	**HashMap $Node**	**Class**	**Concurrent HashMap $Node**
avrora	37,38%	20,31%	72,65%	22,34
fop	28,10%	20,52%	0%	0%
h2	12,46%	0%	0%	0%
jython	20,55%	0%	0%	12,75%
luindex	42,72%	33,36%	79,32%	33,93%
lusearch	22,71%	14,48%	89,78%	15,27%
pmd	19,66%	0%	0%	0%
sunflow	46,15%	14,34%	92,89%	15,05%
tradebeans	30,67%	10,76%	0%	13,49%
xalan	50,77%	19,09%	77,46%	20,37%
average	31,12%	13,29%	41,21%	13,32%
diff	+8,07%	-9,76%	+18,17%	-9,72%

Table 4.8: Average lengths of hot streaks (as a percentage of program lifetime).

Cold Streaks				
Benchmark	**All objects**	**Consistently cold**	**Medium sized objects**	**[B**
avrora	53,90%	0,12%		60,90%
fop	34,03%	0,60%	50%	36,98%
h2	28,21%	2,07%	20,37%	24,51%
jython	31,33%	0,25%	29,16%	32,98%
luindex	64,26%	0,04%		64,13%
lusearch	53,52%	1,42%		55,44%
pmd	28,10%	1,75%		41,88%
sunflow	80,38%	0%	85,72%	81,61%
tradebeans	53,31%	4,36%	0%	63,82%
xalan	77,16%	0,08%		77,93%
average	50,42%	1,07%	37,05%	54,02%
diff	$\pm$ 0	-	-13,37%	+3,60%
Benchmark	**String**	**HashMap $Node**	**Class**	**Concurrent HashMap $Node**
avrora	64,75%	57,40%	38,25%	69,59%
fop	34,77%	48,24%	0%	0%
h2	23,65%	0%	0%	0%
jython	28,52%	0%	0%	30,67%
luindex	64,72%	66,28%	54,55%	60,88%
lusearch	49,96%	80,21%	56,29%	82,04%
pmd	35,89%	0%	0%	0%
sunflow	81,83%	83,75%	65,54%	82,88%
tradebeans	63,47%	67,14%	0%	54,59%
xalan	79,91%	80,92%	57,37%	78,79%
average	52,75%	48,39%	27,20%	45,94%
diff	+2,33%	-2,03%	-23,22%	-4,48%

Table 4.9: Average lengths of cold streaks (as a percentage of program lifetime).

Table 4.10 shows the difference between average hot and cold streak lengths for each benchmark.

Benchmark	Hot	Cold	Diff
pmd	19,42%	28,10%	8,68%
h2	18,66%	28,21%	9,55%
fop	23,92%	34,03%	10,11%
jython	15,57%	31,33%	15,76%
avrora	28,59%	53,90%	25,31%
luindex	37,19%	64,26%	27,07%
lusearch	18,95%	53,52%	34,57%
tradebeans	16,16%	53,31%	37,15%
xalan	33,01%	77,16%	44,15%
sunflow	18,96%	80,38%	61,42%

Table 4.10: Hot and cold streak lengths compared.

From Table 4.8 and Table 4.9, we can see that when grouping all objects, which in the case of the DaCapo benchmark suite is mostly small objects, the average longest hot streak is about 23,04% and the average longest cold streak is about 50,42% of program lifetime.These numbers seem to suggest that cold streaks are longer on average when measured as a percentage of the program lifetime. We can also find outlier results like the benchmarks “sunflow”, where the average longest cold streak is about 80,38% of program lifetime, and “pmd”, where the average longest cold streak is only 28,10% of program lifetime. This could suggest that certain programs would be more or less suited for the ThinGC approach.

For all objects combined, the difference between the highest (luindex, 37,19%) and lowest (jython, 15,57%) ”average longest hot streak” values is about 22% while the difference between the highest (sunflow, 80,38%) and lowest (pmd, 28,10%) ”average longest cold streak” values is about 52%. This seems to suggest that hot streaks in the chosen benchmarks can be expected to fall within a tighter range than cold streaks.

When comparing classes to all objects combined, we see in the ”diff” row of the ”[B” column of Table 4.8 and Table 4.9, that only objects of the ByteArray class have a shorter average longest hot streak (by about 5%) and a longer average longest cold streak (by about 4%). This could suggest that objects of this class are more likely to stay cold than the “average” object.

This can also be seen for example when comparing Figure 4.40 to Figure 4.41.

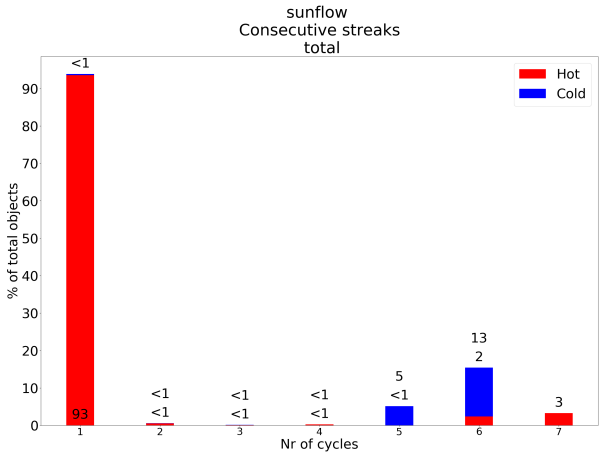

Figure 4.40: Consecutive hot and cold streak lengths of objects in the ”sunflow” benchmark.

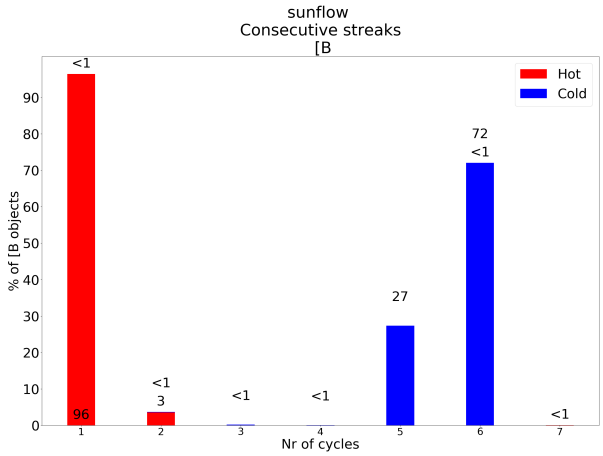

Figure 4.41: Consecutive hot and cold streak lengths of objects of class ByteArray in the ”sunflow” benchmark.

Conversely, when comparing all objects combined to java.lang.Class, we see that this class has a longer average hot streak. It is actually the longest average hot streak among chosen classes at about 41,21% of program lifetime. This class also has a shorter average longest cold streak, the shortest average cold streak among classes at about 27,20% of program lifetime. This could suggest that this class is less likely to stay cold, which can also be seen for example when comparing Figure 4.42 to Figure 4.43.

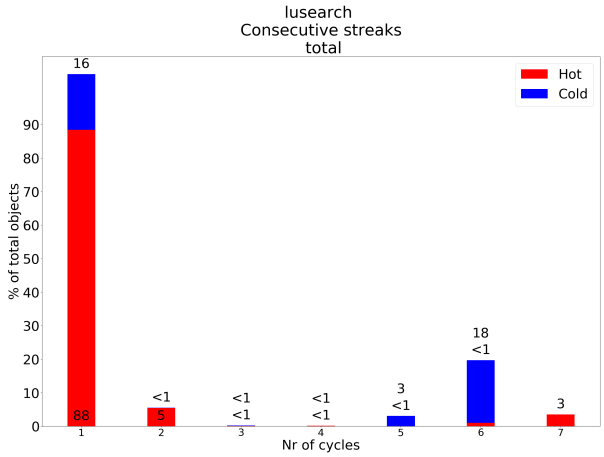

Figure 4.42: Consecutive hot and cold streak lengths of objects in the ”lusearch” benchmark.

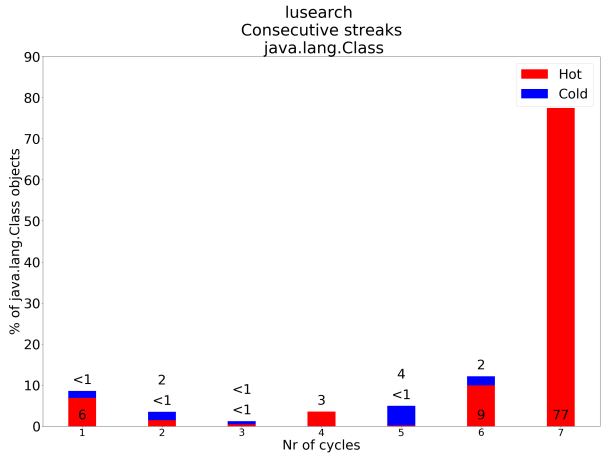

Figure 4.43: Consecutive hot and cold streak lengths of objects of class Class in the ”lusearch” benchmark.

Rating	**Class**	**Avg cold streak**	**Avg hot streak**	**Diff**
1	[B	54%	18%	+36%
2	String	53%	31%	+22%
3	Hashmap	48%	13%	+35%
4	Concurrent	46%	13%	+33%
5	Class	27%	41%	-14%

Table 4.11: Class hot and cold streak lengths compared.

In Table 4.13 we have compared the lengths of the average hot and cold streaks for each class and ranked them by descending average cold streak length. Our theory is that classes with longer average cold streaks and shorter average hot streaks are more likely to stay cold.
This “average consecutive streak” metric would not give credit to short-lived objects which have maintained their consecutive streak throughout their entire life, and would count these streaks as short consecutive streaks. As an alternative to the ”consecutive” metric we therefore provide the following metric.

4.4.2 Mostly & consistently hot/cold objects

As previously stated, we call an object which throughout its lifetime has been either always hot or always cold ”consistently” hot/cold. We now introduce a

”mostly” hot/cold object.

Definition 4.4.1. Mostly hot object. Object with a hot streak lasting from 50 and up to, but not including, 100% of its lifetime. 50%<hot streak<100%.

Definition 4.4.2. Mostly cold object. Object with a cold streak lasting from 50 and up to, but not including, 100% of its lifetime. 50%<cold streak<100%.

The percentages of mostly and consistently hot/cold objects together express the proportion of objects within a program which have had a longest consecutive hot/cold streak lasting longer than 50% of the objects' lifetime. A high percentage of mostly & consistently hot/cold objects would give an assurance of object hot/cold stability throughout the objects' lifetimes. Being aware of an average object hot/cold stability factor could contribute to predicting reheats and avoiding wasteful freezes.
The general limit of 50% was chosen for this thesis as a natural delimiter for a mostly hot/cold object, however, it is possible that measuring alternative (shorter/longer) definitions for mostly hot/cold objects could be interesting in specific cases.

Hot: Mostly & Consistently Combined					
Benchmark	**All obj.**	**All obj. (Consist.)**	**All obj. (Mostly)**	**Medium sz. obj.**	
avrora	28,72%	27,81%	0,91%		
fop	89,08%	77,91%	11,17%	50%	
h2	31,29%	18,33%	12,96%	61,54%	
jython	11,06%	10,49%	0,57%	33,33%	
luindex	73,09%	72,41%	0,68%		
lusearch	76,65%	61,39%	15,26%		
pmd	63,43%	56,82%	6,61%		
sunflow	81,77%	81,45%	0,32%	50%	
tradebeans	27,42%	19,68%	7,74%	100%	
xalan	39,19%	38,42%	0,77%		
average	52,17%	46,47%	5,70%	58,97%	
diff	$\pm$ 0	-	-	+6,80%	
Benchmark	**[B**	**String**	**HashMap $Node**	**Class**	**Concurr. HashMap $Node**
avrora	24,25%	53,23%	23,98%	75%	6,50%
fop	75,68%	87,02%	70,70%	0%	0%
h2	11,11%	12,43%	0%	0%	0%
jython	3,95%	14,72%	0%	0%	1,22%
luindex	54,43%	65,73%	49,82%	98,24%	59,88%
lusearch	71,94%	80,78%	18,67%	91,64%	0%
pmd	72,43%	78,22%	0%	0%	0%
sunflow	0,22%	38,48%	0%	94,04%	5,61%
tradebeans	7,70%	33,55%	1,57%	0%	10,84%
xalan	13,07%	49,30%	55,32%	76,25%	4,83%
average	33,48%	51,35%	22,01%	43,52%	8,89%
diff	-18,69%	-0,82%	-30,16%	-8,65%	-43,28%

Table 4.12: Percentages of mostly and consistently hot objects.

Cold: Mostly & Consistently Combined					
Benchmark	**All obj.**	**All obj. (Consist.)**	**All obj. (Mostly)**	**Medium sz. obj.**	
avrora	71,65%	0,12%	71,53%		
fop	19,14%	0,60%	18,54%	50%	
h2	29,93%	2,07%	27,86%	0%	
jython	88,57%	0,25%	88,32%	83,34%	
luindex	26,04%	0,04%	26%		
lusearch	38,47%	1,42%	37,05%		
pmd	20,07%	1,75%	18,32%		
sunflow	18,19%	0%	18,19%	50%	
tradebeans	75,78%	4,36%	71,42%	0%	
xalan	60,92%	0,08%	60,84%		
average	44,88%	1,07%	43,81%	36,67%	
diff	$\pm$ 0	-	-	-8,21%	
Benchmark	**[B**	**String**	**HashMap $Node**	**Class**	**Concurrent HashMap $Node**
avrora	76,67%	47,02%	76,22%	24,68%	95,02%
fop	50,89%	37,24%	33,41%	0%	0%
h2	28,84%	27,49%	0%	0%	0%
jython	95,43%	85,18%	0%	0%	98,27%
luindex	43,62%	32,82%	49,82%	1,55%	37,14%
lusearch	43,78%	35,04%	85,36%	7,68%	99,95%
pmd	22,96%	16,83%	0%	0%	0%
sunflow	99,61%	61,49%	100%	5,46%	94,36%
tradebeans	91,57%	65,79%	98,69%	0%	84,95%
xalan	87,08%	50,69%	44,68%	20,44%	95,17%
average	64,05%	45,96%	48,82%	5,98%	60,49%
diff	+19,17%	+1,08%	+3,94%	-38,90%	+15,61%

Table 4.13: Percentages of mostly and consistently cold objects.

Across the chosen benchmarks the average percentage of mostly & consistently hot objects is about 52,17% and the cold equivalent is about 44,88%. The average percentage of consistently hot objects is about 46,47% and the cold equivalent is only 1,07%. We know from Table 4.7 that the average lifetime for consistently hot objects is relatively short, about 27% of program cycles. Adjusting for these consistently hot (and cold) objects leaves us with on average 5,7% mostly hot objects and 43,81% mostly cold objects, as can be seen in the ”average” row of the “All obj. (Mostly)” columns of the tables. We conclude that objects are more likely to be consistently hot than consistently cold and more likely to be mostly cold than mostly hot. Hot streaks are more likely to last an entire object lifetime, though it may be relatively short, and cold streaks are more likely to last between 50-100% of object lifetime.
The difference between the highest (”jython” 88,32%) and lowest (”sunflow” 18,19%) benchmark averages for mostly cold objects 4.4.2 is about a 70,13%. The difference between the highest (”lusearch” 15,26%) and lowest (”sunflow” 0,32%) benchmark averages for mostly hot objects 4.4.1 is about 14,94%. Once again, hot streaks seem to be more dependable (within the chosen benchmarks). The average percentage of mostly hot objects can be expected to fall within a tighter range than the average percentage of mostly cold objects.
Medium sized objects seem to be less likely to be mostly & consistently cold and more likely to be mostly & consistently hot. The average percentage of medium sized objects being mostly & consistently cold is 8,2% lower than that of all objects, and the hot equivalent is 6,8% higher than that of all objects.
The ByteArray, HashMap$Node and ConcurrentHashMap$Node classes have lower average percentages of mostly & consistently hot objects and higher average percentages of mostly & consistently cold objects compared to all objects. For example, the ByteArray class has the highest average percentage of mostly & consistently cold objects, about 64,05% and the third lowest average percentage of mostly & consistently hot objects, about 33,48%. This would suggest that objects of these classes are more likely to stay cold during their lifetime than the “average” object which can be seen for example when comparing Figure 4.44 and Figure 4.45 below.

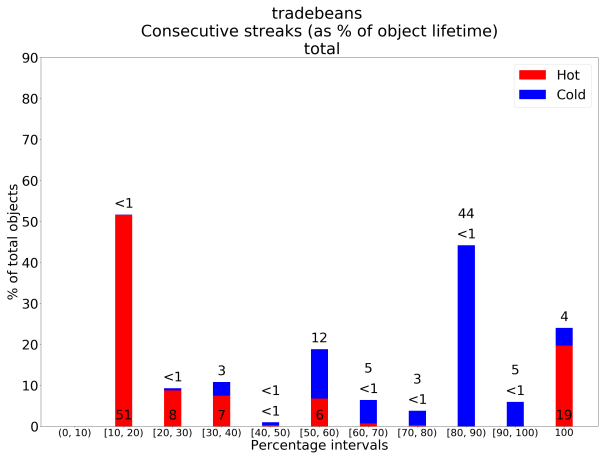

Figure 4.44: Consecutive hot and cold streak lengths of objects in the ”tradebeans” benchmark.

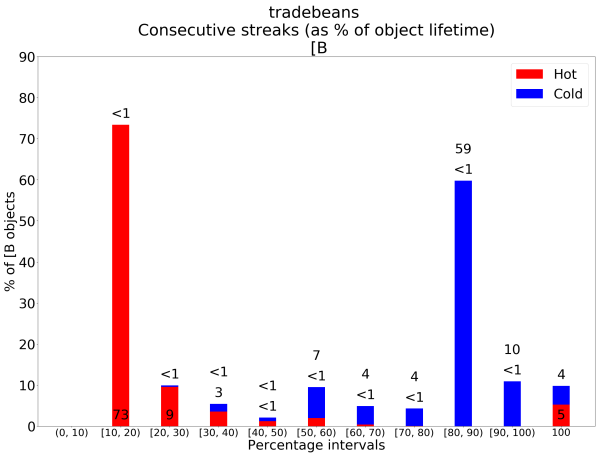

Figure 4.45: Consecutive hot and cold streak lengths of objects of class ByteArray in the ”tradebeans” benchmark.

Conversely, java.lang.Class has the second highest average percentage of mostly & consistently hot objects, about 43,52% and the lowest average percentage of mostly & consistently cold objects, about 5,98%. This suggests that objects of this class, unlike the previously mentioned classes, are unlikely to stay cold during their lifetimes. This can be seen for example when comparing Figure 4.46 and Figure 4.47 below.

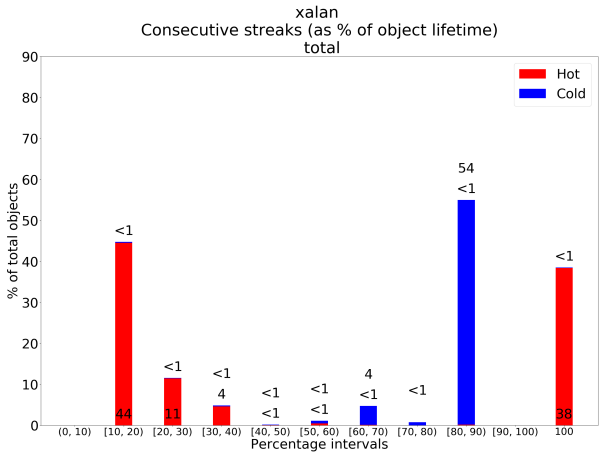

Figure 4.46: Consecutive hot and cold streak lengths of objects in the ”xalan” benchmark.

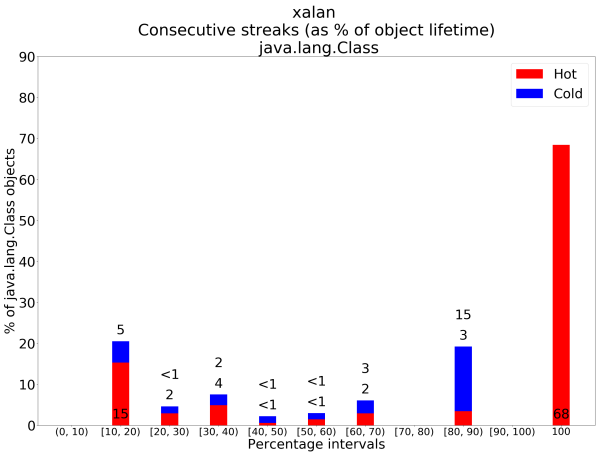

Figure 4.47: Consecutive hot and cold streak lengths of objects of class Class in the ”xalan” benchmark.

This is consistent with the results from the previous "average consecutive streak" metric where the ByteArray, HashMap$Node and concurrentHashMap$Node classes were more likely to have long cold streaks, less likely to have long hot streaks and deemed most likely to stay cold.

Class most likely to stay cold				
Rating	**Class**	**Cold**	**Hot**	**Diff**
1	[B	64%	33%	+31%
2	Concurrent	60%	9%	+51%
3	Hashmap	49%	22%	+27%
4	String	46%	51%	-5%
5	Class	6%	44%	-38%

Table 4.14: Differences between percentages of mostly & consistently hot and cold objects.

In Table 4.14 we have compared the percentages of mostly & consistently hot and cold objects for each class and ranked them by descending cold percentage. We believe classes with a higher mostly & consistently cold percentage and a lower mostly & consistently hot percentage are more likely to stay cold.

4.4.3 Reheats

We have previously plotted reheats and found it uncommon for objects to reheat. We now want to create a metric which expresses the stability of the object. This metric would help us identify objects on a scale from objects which have been reheated a "maximum" possible number of times to objects which have never reheated. We calculate the maximum number of reheats for an object 4.4.3 and express the actual number of reheats as a percentage of this max.

Definition 4.4.3. Maximum number of reheats.
The maximum number of reheats for an object is dependent on the length of the object lifetime and the object hotness in the first GC cycle.
The maximum number of reheats is: $\lfloor \frac{lifetime}{2} \rfloor$ unless the object started hot and has a lifetime of even length, then the maximum number of reheats is: $\lfloor \frac{lifetime}{2} - 1 \rfloor$.

For example, an object which started hot and had a lifetime of 4 cycles could

at most have made one reheat while an object which started cold could at most have made two in the same lifetime.

To prevent the high percentage of consistently hot objects from affecting the average reheat value, and get a more accurate measurement on how eager objects are to reheat once they have become cold, only objects which have in some GC cycle been cold are considered in the calculation of the reheat average.
Table 4.16 is displaying the ten-percent interval into which the average reheat percentage of each benchmark falls, or if the average reheat percentage of the benchmark is 0 or 100%.

Reheats				
Benchmark	**All objects**	**Medium sized objects**	**[B**	**String**
fop	(0%,10%)		(0%,10%)	(0%,10%)
sunflow	[10%,20%)	0%	(0%,10%)	(0%,10%)
lusearch	[50%,60%)	[50%,60%)	[70%,80%)	[70%,80%)
luindex	(0%,10%)	0%	(0%,10%)	(0%,10%)
pmd	(0%,10%)		(0%,10%)	(0%,10%)
xalan	(0%,10%)		(0%,10%)	(0%,10%)
h2	[50%,60%)		[30%,40%)	[30%,40%)
avrora	(0%,10%)	0%	(0%,10%)	(0%,10%)
tradebeans	[10%,20%)	0%	(0%,10%)	(0%,10%)
jython	(0%,10%)		(0%,10%)	(0%,10%)
average	[10%,20%)	[10%,20%)	[10%,20%)	[10%,20%)
Benchmark	**HashMap $Node**	**Class**	**Concurrent HashMap $Node**	
fop	0%	(0%,10%)	(0%,10%)	
sunflow	(0%,10%)	0%	0%	
lusearch	0%	0%	0%	
luindex	0%	0%	(0%,10%)	
pmd	(0%,10%)	[10%,20%)	[10%,20%)	
xalan	0%	[10%,20%)	(0%,10%)	
h2	0%	0%	0%	
avrora	0%	(0%, 10%)	(0%, 10%)	
tradebeans	(0%,10%)	0%	(0%,10%)	
jython	0%	[10%,20%)	0%	
average	(0%,10%)	(0%,10%)	(0%,10%)	

Table 4.15: Ranges of average reheat percentages.

Overall, the reheat averages are very low, usually within the (0, 10) percent range and all benchmarks except “h2” and “pmd” have an average reheat per-

centage below 20%. “h2” and “pmd” however have average reheat percentages within the [50%, 60%) range. For DaCapo we can conclude that in most benchmarks, once an object has become cold there is a low chance of it reheating and it is common for cold objects to stay cold until they die.
The ByteArray and String classes have the highest average reheat values among classes and the java.util.HashMap$Node class has the lowest reheat value, much thanks to many benchmarks having an average reheat value of zero for this class.
The reheat averages mostly seem to follow the conclusions from the previous tables, however one surprising result is that the average reheat value of the ByteArray class is higher than that of java.lang.Class. This could possibly be explained by the higher average lifetime of java.lang.Class objects, 77% vs 50% of program lifetime, which could give java.lang.Class more GC cycles to possibly reheat.

Class least likely to reheat		
Rating	**Class**	**Range**
1	Hashmap	(0%, 10%)
1	Concurrent	(0%, 10%)
1	Class	(0%, 10%)
2	[B	[10%, 20%)
2	String	[10%, 20%)

Table 4.16: A sorting of the classes by average reheat percentage range.

Chapter 5

Discussion and Conclusions

We have presented a method, and suggested metrics, for examining object behaviour in terms of hotness. We have also presented the results from the study of one benchmark suite.
We presented plots where we found reheats to be uncommon and that objects which once had become cold, usually stayed steadily cold until their death, with some benchmarks showing a sudden increase of cold objects in their last GC cycles. We found few (two out of ten) examples of benchmarks where the number of cold objects decreased from one cycle to the next.
We saw that long cold streaks were more common than long hot streaks and that the longest hot streak of objects was often only one GC cycle long. We saw that many objects were consistently hot, but the remaining objects were more likely to be mostly cold than mostly hot. We found that the average lifetime of consistently hot objects was about 32% of program lifetime across benchmarks, significantly lower than the average lifetime of consistently cold objects at about 53% of program lifetime. We presented tables of metrics where we found that the average longest cold streak (50%) was longer than the average longest hot streak (23%), that the average percentage of mostly & consistently hot objects (52%) was higher than the cold streak counterpart (45%) and that the average percentage of mostly cold objects (44%) was significantly higher than the hot streak counterpart (6%).
Since we can generally rely on cold objects staying cold, the efforts to classify objects by hotness and treat cold objects separately could be justified, depending on the added overhead. In the case of this benchmark suite, we can imagine, as a simple starting point, that any overhead need to be recovered on average within the length of the average longest cold streak (50% of program lifetime).

If calculated concurrently, the average hotness information could affect freeze and reheat decisions in ThinGC. For example, knowing the expected reheat frequency and cold streak length could contribute to avoiding disadvantageous freezes. Also, knowing the average or class specific hot and cold streak lengths, ThinGC could anticipate reheats and prefetch objects from cold storage.

From this point it could be interesting to examine if any correlation can be found between allocation site and objects with relatively many reheats in order to predict sub-optimal freeze candidates. Taking inspiration from the methods presented in ”Related Work” section 2.4, to predict long-lived objects which optimises generational garbage collection, such as [50].

We found that the results for our cold metrics like “longest cold streak” and “mostly cold objects”, varied more than the hot streak counterparts. Values for hot streaks seemed to fall within a tighter range than the cold counterparts. This highlights the importance of identifying suitable programs which have longer than average cold cycles in terms of program lifetime, for example “xalan” at 77%. Such programs will likely suit ThinGC especially well, unlike programs like “fop” with 78% consistently hot objects and an average cold streak of 34% of GC cycles.

We also found that objects of certain classes in DaCapo are more or less likely to stay cold. The classes shown were the most frequently used classes in the DaCapo benchmark suite. Further exploration of a larger set of classes from a larger set of programs could confirm our findings. This information could possibly then be used as a baseline for GC tuning.

We believe that our results show distinctly different behaviours of hot and cold objects in this benchmark suite. This information is hopefully useful when developing garbage collection optimisations with object-hotness focus.

Bibliography

[1] Albert Mingkun Yang et al. “ThinGC: Complete Isolation With Marginal Overhead.” Submitted.

[2] Ionel Gog et al. “15th Workshop on Hot Topics in Operating Systems (HotOS XV)”. In: *Broom: Sweeping Out Garbage Collection from Big Data Systems*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog.

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. “Myths and Realities: The Performance Impact of Garbage Collection”. In: *SIGMETRICS Perform. Eval. Rev.* 32.1 (June 2004), pp. 25–36. ISSN: 0163-5999. DOI: 10.1145/1012888.1005693. URL: https://doi-org.focus.lib.kth.se/10.1145/1012888.1005693.

[4] Xianglong Huang et al. “The Garbage Collection Advantage: Improving Program Locality”. In: *SIGPLAN Not.* 39.10 (Oct. 2004), pp. 69–80. ISSN: 0362-1340. DOI: 10.1145/1035292.1028983. URL: https://doi.org/10.1145/1035292.1028983.

[5] Evelyn Duesterwald and Vasanth Bala. “Software profiling for hot path prediction: Less is more”. In: *ACM SIGARCH Computer Architecture News* 28.5 (2000), pp. 202–211.

[6] Wei Xie and Yong Chen. “An adaptive separation-aware FTL for improving the efficiency of garbage collection in SSDs”. In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2014, pp. 552–553.

[7] M.-L. Chiang and R.-C. Chang. “Cleaning policies in mobile computers using flash memory”. In: *Journal of Systems and Software* 48 (3 1999), pp. 213–231. URL: https://doi.org/10.1016/S0164-1212(99)00059-X.

[8] Matthew L Seidl and Benjamin G Zorn. “Predicting references to dynamically allocated objects”. In: *University of Colorado Technical Report* (1997).

[9] Frances S Grodzinsky, Keith Miller, and Marty J Wolf. “Ethical issues in open source software”. In: *Readings in CyberEthics* (2003), pp. 351–366.

[10] Brent Jesiek. “Democratizing software: Open source, the hacker ethic, and beyond”. In: *First Monday* 8.10 (Oct. 2003). DOI: 10.5210/fm.v8i10.1082. URL: https://firstmonday.org/ojs/index.php/fm/article/view/1082.

[11] Bertrand Meyer. “The ethics of free software”. In: *SOFTWARE DEVELOPMENT-SAN FRANCISCO-* 8.3 (2000), pp. 32–36.

[12] Amer Diwan et al. *Energy consumption and garbage collection in low-powered computing*. Tech. rep. 2002.

[13] Colin C Venters et al. “Software sustainability: The modern tower of babel”. In: *CEUR Workshop Proceedings*. Vol. 1216. CEUR. 2014, pp. 7–12.

[14] *Oracle Values and Ethics, Policies and Standards*. https://www.oracle.com/corporate/citizenship/values-ethics.html. Accessed: 2020-01.

[15] *Oracle Code of Ethics and Business Conduct*. https://www.oracle.com/assets/cebc-176732.pdf. Accessed: 2020-01.

[16] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2016.

[17] Paul R Wilson. “Uniprocessor garbage collection techniques”. In: *International Workshop on Memory Management*. Springer. 1992, pp. 1–42.

[18] Taiichi Yuasa. “Real-time garbage collection on general-purpose machines”. In: *Journal of Systems and Software* 11.3 (1990), pp. 181–198.

[19] David Detlefs et al. “Garbage-First Garbage Collection”. In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM ’04. Vancouver, BC, Canada: Association for Computing Machinery, 2004, pp. 37–48. ISBN: 1581139454. DOI: 10.1145/1029873.1029879. URL: https://doi-org.focus.lib.kth.se/10.1145/1029873.1029879.

[20] Christine H. Flood et al. “Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK”. In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ ’16. Lugano, Switzerland: Association for Computing Machinery, 2016. ISBN: 9781450341356. DOI: 10.1145/2972206.2972210. URL: https://doi.org/10.1145/2972206.2972210.

[21] Richard Hudson. *Go GC: Prioritizing low latency and simplicity*. The Go Blog https://blog.golang.org/go15gc. 2015.

[22] Peter Marshall. *Trash talk: the Orinoco garbage collector*. V8 Developers’ Blog https://v8.dev/blog/trash-talk. 2019.

[23] Barry Hayes. “Using Key Object Opportunism to Collect Old Objects”. In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. OOPSLA ’91. Phoenix, Arizona, USA: Association for Computing Machinery, 1991, pp. 33–46. ISBN: 0201554178. DOI: 10.1145/117954.117957. URL: https://doi.org/10.1145/117954.117957.

[24] Per Lidén and Stefan Karlsson. *JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)*. https://openjdk.java.net/jeps/333. Accessed: 2020-01. 2018.

[25] Sylvia Dieckmann and Urs Hölzle. “A study of the allocation behavior of the SPECjvm98 Java benchmarks”. In: *European Conference on Object-Oriented Programming*. Springer. 1999, pp. 92–115.

[26] Shoaib Akram et al. “Managing Hybrid Memories by Predicting Object Write Intensity”. In: *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*. Programming’18 Companion. Nice, France: Association for Computing Machinery, 2018, pp. 75–80. ISBN: 9781450355131. DOI: 10.1145/3191697.3213803. URL: https://doi-org.focus.lib.kth.se/10.1145/3191697.3213803.

[27] Dongchul Park and David HC Du. “Hot data identification for flash-based storage systems using multiple bloom filters”. In: *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE. 2011, pp. 1–11.

[28] Wei Xie and Yong Chen. “An adaptive separation-aware FTL for improving the efficiency of garbage collection in SSDs”. In: *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2014, pp. 552–553.

[29] Wei Xie, Yong Chen, and Philip C. Roth. “A Low-Cost Adaptive Data Separation Method for the Flash Translation Layer of Solid State Drives”. In: *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems*. DISCS ’15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450339933. DOI: 10.1145/2831244.2831250. URL: https://doi-org.focus.lib.kth.se/10.1145/2831244.2831250.

[30] M-L Chiang and R-C Chang. “Cleaning policies in mobile computers using flash memory”. In: *Journal of Systems and Software* 48.3 (1999), pp. 213–231.

[31] Ilhoon Shin. “Hot/cold clustering for page mapping in NAND flash memory”. In: *IEEE Transactions on Consumer Electronics* 57.4 (2011), pp. 1728–1731.

[32] S. Lim, S. Lee, and B. Moon. “FASTer FTL for Enterprise-Class Flash Memory SSDs”. In: *2010 International Workshop on Storage Network Architecture and Parallel I/Os*. May 2010, pp. 3–12. DOI: 10.1109/SNAPI.2010.9.

[33] Changwoo Min et al. “SFS: random write considered harmful in solid state drives.” In: *FAST*. Vol. 12. 2012, pp. 1–16.

[34] David A. Moon. “Garbage Collection in a Large LISP System”. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP ’84. Austin, Texas, USA: Association for Computing Machinery, 1984, pp. 235–246. ISBN: 0897911423. DOI: 10.1145/800055.802040. URL: https://doi-org.focus.lib.kth.se/10.1145/800055.802040.

[35] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. “Effective “Static-Graph” Reorganization to Improve Locality in Garbage-Collected Systems”. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI ’91. Toronto, Ontario, Canada: Association for Computing Machinery, 1991, pp. 177–191. ISBN: 0897914287. DOI: 10.1145/113445.113461. URL: https://doi.org/10.1145/113445.113461.

[36] Xianglong Huang et al. “The Garbage Collection Advantage: Improving Program Locality”. In: *SIGPLAN Not.* 39.10 (Oct. 2004), pp. 69–80. ISSN: 0362-1340. DOI: 10.1145/1035292.1028983. URL: https://doi.org/10.1145/1035292.1028983.

[37] Wen-ke Chen et al. “Profile-Guided Proactive Garbage Collection for Locality Optimization”. In: *SIGPLAN Not.* 41.6 (June 2006), pp. 332–340. ISSN: 0362-1340. DOI: 10.1145/1133255.1134021. URL: https://doi-org.focus.lib.kth.se/10.1145/1133255.1134021.

[38] Trishul M. Chilimbi and James R. Larus. “Using Generational Garbage Collection to Implement Cache-Conscious Data Placement”. In: *SIGPLAN Not.* 34.3 (Oct. 1998), pp. 37–48. ISSN: 0362-1340. DOI: 10.1145/301589.286865. URL: https://doi-org.focus.lib.kth.se/10.1145/301589.286865.

[39] Stephen M. Blackburn et al. “Pretenuring for Java”. In: *SIGPLAN Not.* 36.11 (Oct. 2001), pp. 342–352. ISSN: 0362-1340. DOI: 10.1145/504311.504307. URL: https://doi.org/10.1145/504311.504307.

[40] Timothy L. Harris. “Dynamic Adaptive Pre-Tenuring”. In: *Proceedings of the 2nd International Symposium on Memory Management*. ISMM ’00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 127–136. ISBN: 1581132638. DOI: 10.1145/362422.362476. URL: https://doi.org/10.1145/362422.362476.

[41] Richard E Jones and Chris Ryder. “Garbage collection should be lifetime aware”. In: *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS’2006)* (2006).

[42] Perry Cheng, Robert Harper, and Peter Lee. "Generational Stack Collection and Profile-Driven Pretenuring". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. PLDI '98. Montreal, Quebec, Canada: Association for Computing Machinery, 1998, pp. 162–173. ISBN: 0897919874. DOI: 10.1145/277650.277718. URL: https://doi.org/10.1145/277650.277718.

[43] David A Cohn and Satinder P Singh. "Predicting lifetimes in dynamically allocated memory". In: *Advances in Neural Information Processing Systems*. 1997, pp. 939–945.

[44] Timothy L. Harris. "Dynamic Adaptive Pre-Tenuring". In: *Proceedings of the 2nd International Symposium on Memory Management*. ISMM '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 127–136. ISBN: 1581132638. DOI: 10.1145/362422.362476. URL: https://doi-org.focus.lib.kth.se/10.1145/362422.362476.

[45] Michael S Lam, Paul R Wilson, and Thomas G Moher. "Object type directed garbage collection to improve locality". In: *International Workshop on Memory Management*. Springer. 1992, pp. 404–425.

[46] Hans-J. Boehm. "Reducing Garbage Collector Cache Misses". In: *SIGPLAN Not.* 36.1 (Oct. 2000), pp. 59–64. ISSN: 0362-1340. DOI: 10.1145/362426.362438. URL: https://doi.org/10.1145/362426.362438.

[47] Martin Hirzel et al. "Understanding the Connectivity of Heap Objects". In: *Proceedings of the 3rd International Symposium on Memory Management*. ISMM '02. Berlin, Germany: Association for Computing Machinery, 2002, pp. 36–49. ISBN: 1581135394. DOI: 10.1145/512429.512435. URL: https://doi-org.focus.lib.kth.se/10.1145/512429.512435.

[48] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[49] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. Portland, OR, USA: ACM

Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[50] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. "Dynamic Object Sampling for Pretenuring". In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM '04. Vancouver, BC, Canada: Association for Computing Machinery, 2004, pp. 152–162. ISBN: 1581139454. DOI: 10.1145/1029873.1029892. URL: https://doi.org/10.1145/1029873.1029892.

TRITA -EECS-EX