USING CLASS-LEVEL STATIC PROPERTIES TO PREDICT OBJECT LIFETIMES

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

By

Sébastien Nicolas Marion

May 2009

Abstract (English)

Today, most modern programming languages such as C# or Java use an automatic memory management system also known as a Garbage Collector (GC). Over the course of program execution, new objects are allocated in memory, and some older objects become unreachable (die). In order for the program to keep running, it becomes necessary to free the memory of dead objects; this task is performed periodically by the GC.

Research has shown that most objects die young and as a result, generational collectors have become very popular over the years. Yet, these algorithms are not good at handling long-lived objects. Typically, long-lived objects would first be allocated in the *nursery* space and be promoted (copied) to an older generation after surviving a garbage collection, hence wasting precious time.

By allocating long-lived and immortal objects directly into infrequently or never collected regions, pretenuring can reduce garbage collection costs significantly. Current state of the art methodology to predict object lifetime involves off-line profiling combined with a simple, heuristic classification. Profiling is slow (can take days), requires gathering gigabytes of data that need to be analysed (can take hours), and needs to be repeated for every previously unseen program.

This thesis explores the space of lifetime predictions and shows how object lifetimes can be predicted accurately and quickly using simple program characteristics gathered within minutes. Following an innovative methodology introduced in this thesis, object lifetime predictions are fed into a specifically modified Java virtual machine. Performance tests show gains in GC times of as much as 77% for the "SPEC jvm98" benchmarks, against a generational copying collector.

Résumé (francais)

Aujourd'hui, la majorité des langages de programmations modernes tels que C# ou Java font appel à un "ramasse-miettes" (GC) pour gérer l'utilisation de la mémoire. Durant l'exécution d'un programme, de nouveaux objets sont créés, et d'anciens objets deviennent inaccessibles (meurent). Afin que le programme puisse continuer son exécution, il est parfois nécessaire de libérer la mémoire de tous les objets morts. Cette tâche est accomplie par le GC.

Dans la mesure où la plupart des objets meurent jeune, les GC générationnels sont devenus, au fil des ans, très populaires. Pourtant, ces algorithmes ne gèrent pas efficacement les objets ayant un long cycle de vie. Tout nouvel objet est d'abord créé dans la *maternité* (*nursery* en anglais). Si ce dernier survi à une phase de GC, il est alors copié dans l'espace *mature*, moins fréquemment collecté.

En créant les objets à long cycle de vie directement dans l'espace *mature*, le procédé dit de *pretenuring* permet de réduire considérablement le temps imparti au GC. La méthode la plus efficace pour prédire la durée de vie d'un objet consiste à enregistrer la trace du programme à optimiser, ce qui peut prendre plusieurs jours et générer plusieurs giga-octets de données qui doivent alors être analysés. De plus, cette analyse doit être faite pour chaque nouveau programme.

Cette thèse démontre comment la durée de vie d'objets jamais rencontrés auparavant peut être prédite précisément en quelques minutes en exploitant de simples caractéristiques statiques propres à chaque programme. Ces prédictions, sont alors utilisées dans une machine virtuelle spécialement modifiée.

Par rapport à un GC copieur générationnel, nous enregistrons des gains de performance pouvant atteindre jusqu'à 77% pour des programmes de" SPEC jvm98".

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Mr. Richard Jones, who always guided me and pushed me throughout this long journey. He spent countless hours answering my questions and reviewing my work. He forced the best out of me through our weekly supervision meetings.

I would like to sincerely thank Dr. Chris Ryder, who helped me countless times during my Ph.D and who allowed us to use his experimental JVM.

I wish to thank Dr. Jeremy Singer with who it was a real pleasure collaborating.

I am very grateful to Dr. Alex Freitas for all the help he provided me with regards to data-mining as well as Dr. Gisele Pappa and Fernando E. B. Otero.

A warm thank you to Hsuen Pei Ting for helping me with some statistics.

I would like to express my sincere thanks to Dr. Carolina Filippini, Pulitha Liynagama and Dr. Philipp Mohr for being supportive of my Ph.D. even during critical business times at Comufy.

A warm thank you to all my friends, housemates and Ph.D. fellows who shared this long journey with me, in particular Axel, Carlos, Caro, Damian, Damjee, Erick, Fernando, Gift, Gisele, Kitty, Leishi, Lingfang, Lukas, Martino, Mudassar, Myo, Nick, Olaf, Phil (s), Patrick, Poul, Puli, Rodolfo, Sabrina and Tooba.

My loving thanks to my girlfriend, Ann, whose support and understanding helped me carry on.

Enfin, je voudrais particulièrement remercier toute ma famille et en particulier mes parents, Gérard et Laurence, qui m'ont toujours soutenu pendant ces longues années. Au creux de la vague, quand je perdais espoir, ils étaient toujours là et continuaient à croire en moi. Merci, je vous dois tant.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

From personal computers to mobile phones, from global positioning systems to microwaves, we encounter processors, memory, and software. The never ending increasing demand for new and more complex programs combined with regular hardware improvements has had important repercussions in terms of memory and software architecture. Top of the range personal computers now come with as much as 32 GB of main memory, and are powered by processors made of up to eight cores. IBM's Roadrunner [13], the current most powerful supercomputer in the world, built for the U.S. Department of Energy, manages as much as 103.6 TiB of main memory and is powered by 12,960 IBM PowerXCell 8i processors and 6,480 AMD Opteron dual-core processors for a peak throughput of 1.7 petaflops.

In order to adapt to the growing complexity of computer architecture and software, programming languages are evolving. Strong emphasis is put on reusable components at all levels, from the core of the operating system to the end-user software. To facilitate this process, the concept of Object Oriented (OO) programming was introduced in 1967 in a programming language known as Simula 67 [114]. In 1979, Bjarne Stroustrup at Bell Labs gave a new dimension to OO, by developing the language *C with Classes*, later renamed to *C++* [150], bringing object oriented paradigm to the popular C programming language. Today, OO has become extremely popular and many languages such as C++, C#, Eiffel, PHP, Python and Ruby support object oriented concepts.

However, as a natural consequence of growing software complexity, manual memory management has become increasingly more difficult, and it is sometimes impossible to determine prior to execution the lifetime of certain objects or data structures in highly complex systems. For this reason, and despite the rapid growth in memory sizes, automatic memory management, also known as Garbage Collection (GC) is still an important area of research today. GC was introduced in Lisp in 1958 [16] to relieve the programer from the fastidious and error-prone task of manually managing memory resources. While programmers still have to request memory from the runtime in order to create new objects, they do not have to free it, allowing them to concentrate on higher level problems, and to make more efficient use of their time. When enough memory space is no longer available to allocate new objects, a garbage collection is triggered and any object that cannot be used by the program in future is reclaimed. The space that has been reclaimed is then reused to allocate new objects, in order to keep the program running safely. Today, GC is a very popular feature and is implemented in most modern programming languages such as Eiffel, Smalltalk, Ruby, Python, D, Java and C#.

Unfortunately, automatic memory management does not come free and programs developed in a language that allows garbage collection often have associated overheads. This is because the execution environment, in which the program runs, has to perform periodic garbage collections in order to ensure that the program has enough memory available to work correctly. As a result, the overall time needed to execute a program is extended, and occasional pause-times, needed to perform the task of garbage collection, are introduced.

1.1 Context Of This Thesis

The role of any garbage collector is to reclaim the memory space used by dead objects in order for it to be reused. Ideally, a garbage collector should do only this. Unfortunately, garbage collectors do not know the lifetime of objects ahead of

time, and therefore are unable to optimise automatically the placement of objects in such a way that dead objects can be reclaimed all at once. Hence, garbage collectors often have to perform an expensive scanning process to determine which objects are dead, or have to copy non-dead objects from one part of memory to another.

Researching on the lifetime of objects, Ungar developed in 1984 the *weak generational hypothesis*, based on the observation that "most objects die young" [163]. To take advantage of this discovery, generational garbage collectors have been introduced, and remain a very popular choice today. Generational garbage collectors divide the heap into two or more spaces. Objects are allocated into a first space called the *nursery*, where most of them will die. Objects surviving a nursery GC are then *promoted* (or copied) into the second space (also known as the *mature space*), where they are given more time to die (see Chapter 2 for more information on generational collectors). If a nursery GC can not reclaim enough space to resume normal activity, then a full heap GC is triggered, where both nursery and mature spaces are collected.

At each nursery GC, only live objects and objects referenced from the mature space have to be traced and copied. Conversely, not processing dead objects is cheap since they are not traced, nor copied.

Unfortunately, because not all objects die young, generational collectors do not perform well with certain programs [12, 26, 22]. Generational GCs do not handle objects with long lifetimes well because they may be processed several times: promoted to the mature space, and then copied during each full heap GC until they die. For this reason, some memory management implementations such as MMTk [21] and HotSpot [113] treat objects that remain live during the entire execution of the program specially. These objects that never die are often referred to as *immortal* objects, and the dedicated space in which they are allocated is never processed, thereby saving processing time.

Because processing (tracing and copying) live objects is expensive, we do not want to process objects unnecessarily, but we would like to process them as soon

as they die [131] to avoid wasting space. A popular technique for reclaiming dead objects as soon as they die consists of grouping objects with similar lifetimes together. Several techniques have been explored to do so, such as offline processing and pretenuring [14, 46, 171, 88, 26, 22]; region inference in functional languages to get a stack discipline [160, 155, 166, 17, 43, 44, 68, 67, 69]; pointer analysis with Connectivity Based Garbage Collection (CBGC) [80, 79, 66]. The above mentioned techniques are detailed briefly below. For a further description, please refer to Chapter 2.

Röjemo and Runciman studied the *lag* (time between the creation of an object and its first use) and *drag* (time between the last use of an object and its deallocation) of objects throughout their lifetime [131, 134]. Their research revealed that many objects are allocated long before they are first used, while some are simply never used (*void*). Using this information, it should be possible to allocate objects in such a manner as to reduce the effect of lag, drag and void in heap memory.

Tofte and Talpin provide region-based memory management [160, 161, 162, 158, 154, 155, 157, 159, 156] where all values are stored in a so-called stack of regions at runtime. Regions can grow dynamically to accommodate recursive types, such as lists and trees. Regions and objects to put in each are identified offline, using a pointer analysis: a static code analysis technique that establishes which pointers, or heap references, can point to which variables. By using a stack of regions, less objects need to be allocated in the heap, hence reducing the overhead of a garbage collector. Also, regions are reclaimed as a whole, rather than individual objects. However, the price to pay for region-based memory management is added complexity for the programmer who needs to get familiar with the concept and learn the syntax necessary for its use.

To overcome this issue, Rugina and Cherem [41] propose a static analysis that automatically transforms standard Java code to add region specific code. Their analyses indicates that this approach is able to place a large fraction of the objects in regions or on stack. But while their system yields significant absolute

memory savings for several benchmarks when compared to not running a GC, it increases the space requirements in most cases when compared to a garbage collected system.

On the basis that connected objects share similar lifetimes [79], Guyer and McKinley [66] seek to colocate them in the same space. Their scheme, called Connectivity Based GC (CBGC), combines static analysis, with a specialised allocator, that places the new object in the same space as the object to which it is connected. As well as reducing copying, colocation also reduces pressure on the write barrier (see Section 2.1.3). Experiments with jvm98 show that GC time can be reduced by up to 75%.

In order to avoid processing objects that are not dead, Cheng et al. [39], and Blackburn et al. [26, 22] record and analyse the objects lifetime behaviour of a program offline, before running it again with specific allocation. By knowing ahead of time how long an object is going to live, they allocate objects with a shorter life-span together in the nursery, and objects with longer life-span in the mature space or in an immortal space which is never collected. This technique, called *pretenuring*, reduces the amount of objects that survive the garbage collection of the nursery and have to be promoted (copied) to the mature space. Furthermore, by using an immortal space, they avoid processing immortal objects completely.

Cheng et al. make offline pretenuring decisions based on the fraction of objects allocated at a specific site that survive a minor GC.

Blackburn et al. [26, 22] improve this approach by normalising the lifetime of objects against the maximum volume of objects live at any time in the run of the program. After recording and analyse tracefiles, they categorize the lifetime of each allocated object as either *short-lived*, *long-lived* or *immortal*. From this information, they calculate which allocations sites allocate mostly short-lived, long-lived and immortal objects. Once lifetimes have been determined, a second run of the program is performed, where objects allocated at mostly long-lived sites are allocated directly into the mature space, and objects allocated at mostly immortal sites are allocated directly into the immortal space. Their scheme shows

impressive GC-time improvements of between 40% to 70% on average for most heap configurations.

But while pretenuring can reduce the time spent in GC, it suffers from important drawbacks, such as the necessity to perform offline recording and analysis of program traces prior to running that same program. This task can easily take days, and is very impractical for the end-user.

1.2 Aim Of This Thesis

Knowing ahead of execution time the lifetime of objects can be useful. This thesis revolves around the belief that the understanding of object lifetimes and their behaviours throughout the execution of the program holds a key to major time-saving optimisations.

With an accurate knowledge of object lifetimes, we can avoid repeatedly processing objects, and reclaim them soon after they die. Grouping objects with similar lifetimes can be achieved using region inference, colocation (CBGC), or pretenuring.

In this thesis, we focus on making pretenuring more practical to the end-user. This thesis explore ways by which the lifetime of objects can be predicted ahead of time, without the need for a resource consuming recording and analysis of program traces. This thesis also demonstrates how the analysis of class-level static properties can be done in seconds, and yet reduces the time spent in garbage collection by as much as 77%.

1.3 Approach

This section introduces the notion of *predictors*, which identify specific class-level related features and are the basis of the object lifetime prediction system. This thesis explores two possible predictors, namely Chidamber and Kemerer's [42] software metrics (CK metrics), and Micro-Patterns [62].

CK metrics are some of the most commonly used software metrics in the industry, and are generally used to assess code quality. Micro-patterns on the other hand are similar to design-patterns, but are mechanically recognisable and target class-level features rather than interaction between classes.

This thesis shows how to record and analyse program traces, and match each allocation site[1] with a lifetime prediction and a set of predictors.

By applying data-mining techniques to this information, rules are generated where the presence or absence of certain predictors indicates a specific lifetime. These rules are stored in a knowledge bank, which needs to be gathered only once, typically by the researcher or the vendor. Once this knowledge bank has been constituted, the end-user can reuse these rules to predict the lifetime of objects within programs never encountered before.

Thanks to the knowledge bank, a quick analysis of a previously unknown program is sufficient to uncover its predictors and generate lifetime predictions by matching these predictors against the rules stored in the knowledge bank.

The advice we generate is written into an advice file which is then loaded by a specially modified virtual machine capable of taking advantage of such prediction to allocate objects in the most appropriate space and ultimately reduce the time spent in GC.

1.4 Contributions

This thesis makes the following contributions to the field of garbage collection:

- It is shown how data mining can be used to generate accurate lifetime predictions.
- It is shown that the intent of the programmer can be captured by the use of predictors.

[1] Every point in the program at which the allocation of memory occurs.

- It is shown that certain predictors exist whose presence strongly correlates with a particular lifetime category, that can be used to predict object lifetimes.

- A knowledge bank is used to obtain predictions for the lifetimes of objects allocated by individual sites; this lifetime advice is generated quickly.

Furthermore, reductions in garbage collection time of up to 77% for the jvm98 benchmarks are obtained against a generational copying collector.

1.5 Structure Of This Thesis

The rest of this thesis is organised in 7 chapters.

Chapter 2 covers the background necessary for a good understanding of this thesis. It explores the different garbage collection algorithms that have been developed over time as well their strengths and weaknesses. It then reviews work relevant to object lifetime prediction and pretenuring.

Chapter 3 details the methodology used to carry out experiments. In particular, it describes the approach to object lifetime classification, as well as how program trace files were produced and analysed . It then reviews the use of data-mining to create an object lifetime knowledge bank.

Chapter 4 describes implementation in detail and the way that advice files are loaded and treated by the JVM. It also describes the modifications made to the JVM to support the experiments.

Chapter 5 introduces software metrics and shows how the CK metrics were used to drive object lifetime prediction. It presents and discusses the performance achieved in our modified JVM when using advice generated via the use of the CK metrics.

Chapter 6 explains the notion of micro-patterns and shows how we successfully employed them to predict object lifetimes. It presents several sets of experimental results and discusses performance obtained in this case.

Finally, Chapter 7 presents a summary of this thesis, the results achieved and discusses the potential for future work.

Chapter 2

Background

This chapter comprises of four parts. The first part is a review of the main garbage collection algorithms such as reference counting, basic tracing collectors, and generational garbage collectors. The second part is a detailed review of *pretenuring*, an optimisation technique based on the prediction of object lifetimes on which the whole of this thesis is based. The third part summarises garbage collection techniques and present the conclusions in this domain. The final part gives the reader an introduction to data-mining, which is used in this thesis.

2.1 Main GC Algorithms

The purpose of any garbage collector is to collect pieces of data that are no longer used by the program, in order for the space they occupy to be reused. In this context, one such piece of data can interchangeably be called a *cell*, a *node*, or an *object*. When an object is no longer used by the program, it is considered *dead*. Conversely, an object that is still used by the program is considered *live*.

Computer programs regularly manipulate registers, the program stack and global variables. Any such location holding references to a heap object is considered part of the *root set*.

An object in the heap is considered live if it is directly reachable from roots, or by following a chain of pointers from other live objects. The 'points-to' relation

is defined as $\rightarrow$. For any object M and any heap object N, $M \rightarrow N$ if and only if M holds a reference to N. The set of live objects is defined as the *transitive referential closure* of the set of roots. Jones and Lins [93] define the set of live objects as:

$$live = \{N \in Objects \mid (\exists r \in Roots.r \rightarrow N) \vee (\exists M \in live.M \rightarrow N)\} \quad (1)$$

At any point during execution, the live set can be described as the actual set of objects potentially accessible by the program. However, there may exist amongst this set, some objects that a data-flow analysis by an optimising compiler would reveal dead. The above definition therefore constructs an over-estimate of the live set. For example, there may exist in a register a pointer that is no longer valid, but has not yet been cleared in order to save computation time.

During computation, objects die because of mutations in the graph of reachable objects introduced by pointer updates. For this reason, the running program is often referred to as the *mutator* (or *scavenger*) because as far as garbage collection is concerned, the only thing it does is mutate the graph of reachable objects.

Throughout this thesis, the *lifetime* of an object is referred to as the time elapsed between when it was first allocated, and when it was last reachable. Time however can be measured in different ways. Using wall-clock time may seem like the obvious way of measuring time, but it has a few weaknesses when applied to computers and garbage collection. For instance, wall-clock time would vary based on the particular implementation, environment and machine the tests are run on. Instead, a standard way of measuring time for garbage collection is to measure time in bytes of heap allocated. Measuring time in bytes gives a standard machine-independent way for measuring objects lifetimes.

Note that this is not a perfect way of measuring time because it is indeterminate in the presence of threads. Also, a particular lifetime measurement problem arises in the Jikes RVM [1], a Java Virtual Machine (JVM) written in Java, formally known as the *Jalapeño* virtual machine. This JVM, on which experiments

are performed, comes with a "Just In Time" (JIT) compiler which compiles and optimises Java bytecode at run-time. This involves the allocation of data to perform compilation, therefore artificially increasing the lifetimes of user program objects even if all user threads are stopped.

This thesis focuses on mechanisms to accurately predict object lifetimes in order to improve garbage collector efficiency when reclaiming dead objects. The first algorithm reviewed is reference counting because it is ideal in terms of object lifetime management — this algorithm allows (theoretically) for objects to be deallocated as soon as they become unreachable[1].

2.1.1 Reference Counting

Reference counting was first introduced by Collins in 1960 [47] and remains used in some implementations today, such as real-time systems and libraries such as boost [31] for C++. This algorithm has been the primary choice by many early systems which could not tolerate pause-times, such as Modula-2+, SISAL, awk and perl [63, 132, 35, 50]. In the past few years, a resurgence of interest in reference counting has occurred, and some recent papers readdress this technique, partly in the context of multi-core applicability [103, 104, 49, 108, 7, 8, 10, 24, 121, 120, 122].

In this algorithm, every cell in the heap keeps a counter in its header called a *reference count* (RC) which is used to store the number of objects pointing to the cell. Every time a reference to the object is created, the RC is incremented, and every time a reference to an object is removed, the RC of the pointed to object is decremented. When the RC of a cell drops down to zero, no pointers to the cell remain. The cell is therefore considered garbage and can as such be recycled. When a cell X is deleted, any pointers originating from X must also be deleted, and RCs of cells originally pointed to by X must be decreased. This operation is recursive and if any cell's reference count becomes 0, then it is deleted in the same manner.

[1] Unless in the case of a cyclic reference (see Section 2.1.1).

The principal advantage of this algorithm is that the overhead associated with updating reference counts is distributed throughout the program, making it eligible for real-time systems. This algorithm does not incur pause-time, unless reclaiming large linked data-structures. In that respect, it very much differs from a *stop the world* collector such as a *tracing collector* (see Section 2.1.2).

However, reference counting has some drawbacks. For instance, every time a reference is created or destroyed, the reference counters of its old and new targets need to be updated to reflect the change. As a consequence, maintaining reference counts is computationally intensive, leading to comparatively bad program throughput [24]. In multi-threaded environments, reference counting can be a problem since each modification of the reference count would typically require synchronisation. Finally, the major drawback of reference counting, first noted by McBeth [110], is its inability to naturally collect cyclic structures. For instance, if two cells A and B reference each other and the last external pointer to each cell has been destroyed, then each of them will still have a reference count of 1 and so will not be collected. This causes memory leaks which will reduce the available memory and can potentially lead to an out of memory condition.

Several approaches have been taken to tackle this problem, such as using a hybrid-algorithm [168] where most cells would be handled by reference-counting, but occasionally a *mark-sweep* collector will be invoked to collect cyclic references.

The overhead of maintaining reference counts is high. Every pointer update requires adjustments of reference counts for both the old and the new target cells, making naive reference counting unfeasible for real systems.

Loading and updating local variables located in the stack or in registers whenever necessary is very expensive. To tackle this problem Deutsch and Bobrow [52] propose a scheme, called *deferred reference counting*, in which they only store the reference counts of cells referenced by other cells in the heap. Local variables in the stack are not reference counted, saving computation time. All cells with a reference count of 0 are added to a Zero Count Table (ZCT). If a cell referenced by the ZCT becomes reachable by another cell in the heap, its original reference

is removed from the ZCT. The periodical finding of dead cells is done by checking what cells referenced by the ZCT are not referenced by any local variables in the stack. This algorithm is interesting because a lot of time can be saved by not reference counting the stack. Furthermore, by deferring some of the processing to later, this algorithm has a much faster throughput than naive reference counting. This is possible because all garbage cells can be identified by scanning the stack and the ZCT. Its downside, however, is that by not reference counting variables in the stack, it is no longer possible to reclaim cells as soon as they become unreachable.

Bobrow [27] on the other hand has proposed that the programmer should assign all nodes to groups, and that only groups should be reference counted. This way, cyclic structures inside a group would still be reclaimed as soon as the group is reclaimed. The downside of this technique is that it requires a certain programming paradigm or relies on programmer intervention.

Brownbridge [32] introduced the idea of differentiating cycle-closing pointers from other pointers, but Salkild [136] identified a flaw in Brownbridge's algorithm, and fixing the algorithm would introduce termination problems. Pepels et al. [124] and Thompson et al. [153] found new ways of handling the termination of this algorithm, but their implementations were prohibitively computation intensive.

Extending over previous work by Martinez et al. [109], Lins [107] proposes an extension to the classic reference counting algorithm. Every time a pointer to a cell with multiple references is deleted, a reference to these cells is added onto a queue. Most cells with multiple references are reclaimed naturally when their reference count comes down to zero (i.e: they are not cyclic structures). Whenever the free-list becomes empty, or when the queue is full, a mark-scan phase using the queue as the root set is performed to reclaim the space used by cyclic structures. Unfortunately, this algorithm considers roots one at a time, performing the reference count updates for that root before processing the next root. In other words, a complete scan will be performed from each root. Bacon and Rajan address this problem by adding an extra flag to every object to ensure

that objects or structures are not visited more than once [10], leading to a much improved throughput.

Levanoni and Petrank [103] have devised a scheme where they use reference counting for concurrent garbage collection. In this scheme, pointer updates and creation of objects require no synchronisation overhead. Program threads are never stopped simultaneously to cooperate with the collector, and each program thread cooperates with the garbage collector at its own pace. Thread pauses are kept very short and infrequent. Their *sliding views* algorithm uses local *history buffers* to record pointer changes since the last time the thread cooperated with the GC. During each thread cooperation, the local history buffer associated with the thread is processed and reference counts are updated. Periodically, a tracing collector is used to reclaim cyclic structures. Several papers have been published based on this technique in the past few years [105, 121, 122].

Blackburn and McKinley's ulterior reference counting [24] combines deferred reference counting with a copying nursery, observing that the majority of pointer mutations occur in young objects. Their collector is divided into two spaces, the former being collected more frequently than the latter[2]. The frequently collected space is small, and is collected using a copying collector (see Section 2.1.2). The infrequently collected space, which is larger, is collected using a deferred reference counting algorithm, in order to avoid pause-times introduced by tracing the entire space. This algorithm achieves throughput comparable with the fastest generational copying collectors (see Section 2.1.3) with the low bounded pause times of reference counting. It is interesting with regards to this thesis because it makes efficient use of object lifetimes (using a generational GC), and reduces pause-times by using a deferred reference counting algorithm to manage the infrequently collected space.

As was seen in this section, reference counting is attractive because garbage collection is interleaved with the program's execution, hence reducing GC related pauses. But reference counting has drawbacks. First, tracking pointer changes

[2]More precisely, they use a *generational collector* which is detailed in Section 2.1.3.

requires the use of a *write-barrier*, which by performing extra checks at each pointer write adds a constant overhead to the program (see Section 2.1.3 for a discussion about write-barriers). Second, since the overhead of reclaiming objects as soon as they become unreachable would in practice be too high because of the number of instructions required to update the reference counts of both the old and new target cells, techniques such as deferred reference counting have to be used. This in turn implies that high-performance reference counting garbage collectors *do not* collect garbage as soon as it becomes unreachable. Third, because reference counting is unable to reclaim cyclic structures naturally, an extra mechanism needs to be used from time to time, often a tracing collector.

With an oracle capable of accurately predicting lifetime, one could have an ideal GC algorithm that has reduced pauses times like reference counting does, but without the overhead of keeping track of reference counts at each pointer write, hence increasing throughput. This thesis reveals the design of a lifetime prediction system which is both cheap and accurate.

2.1.2 Tracing Collectors

This section reviews the fundamentals of tracing collectors.

In tracing collectors, *GC roots* are used as a starting point for garbage collection. They are usually made of temporary variables on the stack (of any thread) and static variables. Any object (or cell) reachable from the GC roots by traversing the graph of connected objects is considered *reachable* or *live*. In contrast, a cell is considered *dead* if it is unreachable.

During the execution of a program, references between objects are created, and a graph linking live objects together is produced. At every GC, tracing collectors walk the graph of reachable objects to identify live objects. All non-live objects are considered dead, and are therefore reclaimed.

Unlike naive reference counting, where the overhead is distributed throughout the program, collections in a tracing collector are triggered once a certain memory threshold is reached, typically when no more memory is available. For *stop the*

world collectors (as opposed to *concurrent* collectors), all running threads are stopped while the collection is performed in order to prevent the graph of live objects from being modified while the garbage collector is running. Failing to stop every thread before collection could cause an object to be incorrectly reclaimed if it was not identified as live during the tracing phase.

A major improvement over reference counting is that by walking a graph of live objects, cyclic references are naturally dealt with, but on the other-hand stopping active threads and performing GC induces pauses in the program.

This thesis introduces below *Mark and Sweep* collection and *Copying* collection (also referred to as semi-space collection), as the two main tracing mechanisms used today. Tracing algorithms form the underlying basis of most leading edge garbage collectors including those such as *generational* collectors (see Section 2.1.3).

Mark And Sweep collector

Mark and sweep garbage collection, introduced in 1958 by McCarthy [112], was the first automatic memory management algorithm discovered. Despite its age, mark and sweep collection still performs well in certain cases and is still used in a variety of different contexts. Under this scheme, dead objects are left in memory until no more memory can be allocated, at which point the next allocation request will be suspended until the mark-sweep algorithm has reclaimed all dead objects.

Mark and sweep traverses the graph of reachable objects from the roots by following pointers. It identifies all reachable (hence live) cells in a phase known as *marking.* This operation usually (but not always) requires dedicated extra room in each object header to tag objects that the marking phase detected as live. There are a variety of techniques to perform the marking phase, and the most popular techniques are reviewed later in this section.

At the end of the marking phase, all reachable objects are *marked* and all unmarked objects are considered dead. Dead objects are returned to the pool of free cells in a phase called *sweeping.* This is usually achieved by walking the heap

linearly from bottom to top and returns to the pool of free cells every unmarked cell (dead). If the sweeping phase fails to reclaim enough memory, the heap must be expanded or the application would have to be aborted.

The pool of free cells can be managed by different data-structures which all have their strengths and weaknesses. The following paragraph reviews briefly some of these implementations.

Using a mark stack In order to perform marking, researchers often use iterative loops coupled with auxiliary data-sructures. For instance, one can use an auxiliary stack to hold pointers to live nodes that have not yet been visited. This stack would initially contain pointers to the root nodes. During the mark phase, nodes are repeatedly popped from the stack, and any unmarked children are marked and pushed onto the stack.

Using a mark stack, time and space requirements are made explicit. The maximum depth of the marking stack depends on the size of the longest path that has to be traced. Some studies have focused on trying to reduce the depth of the stack [45, 102, 29]. Others have focused on detecting and recovering from stack overflows [5, 99, 29].

Processors often provide instructions that allow data to be *prefetched* into the cache ahead of time, in order to reduce the delay associated with retrieving it from RAM. New studies have investigated how prefetching objects located in the stack can reduce cache misses [28, 40, 61].

In a 1991 study, Zorn explores the effect of garbage collection on cache performance and found that garbage collector activity can have a significant impact on cache performance. For instance, he found that on certain processors, a generational GC with a copy mature space has a cache-miss rate up to four times higher than a generational GC with a mark-sweep mature space.

Pointer reversal Using a mark-stack requires a potentially unbounded amount of space. While research has investigated how to reduce the space requirement

or recover from a stack overflow, *pointer reversal* [137, 99] can perform marking without requiring any extra space at all. Rather than using an auxiliary data-structure or some extra header bits to store the visited nodes, *pointer reversal* simply stores back-pointers to the previously marked node in a pointer field. As the marking phase traverses the graph recursively, every pointer followed is reversed by putting the address of the parent cell in one of the pointer fields of the cell being examined. As the recursion unwinds, the original values of all pointer fields are restored. Unfortunately, this technique is very expensive because of the number of pointer writes required to traverse the graph, considerably slowing down the marking phase. Furthermore, pointer reversal does not work well with incremental GC. For these reasons, very few implementations use pointer reversal today.

Bitmap marking This technique takes away the need for an extra-bit in the header of each object by using a dedicated bitmap table to store object mark-bits. More specifically the bitmap table associates a bit for each object address in the heap that may contain the start of an object. Instead of using a simple linear array of bits, Boehm and Weiser suggest that this structure could be further improved by using separate bitmap tables for each different type of objects [29]. A hash table or search tree would then be used to access the information.Separate bitmap tables also have the advantage that they cope well with a very fragmented heap and that not every location in the heap requires a mark-bit.

Using a bitmap has repercussions on virtual memory and RAM: if the bitmap is small enough to fit in cache, then accessing a mark-bit does not require loading the associated object into memory, so reading and writing mark-bits would not incur cache misses.The downside to this implementation however, is that accessing the bit associated with each object through the bitmap is more expensive than accessing it through the object's header [172].

Lazy sweeping One way to reduce the stop the world time consists of doing a fixed amount of sweeping at each allocation [87]. This semi-incremental technique, known as *lazy sweeping* was introduced by Hughes as a mechanism for reducing paging and pause times. Rather than reclaiming all free cells atomically, lazy sweeping reclaims free cells on demand (by the allocator). Unlike a classical mark-sweep garbage collector, the pause-time involved is proportional to the amount of the heap in use, not to the size of the heap. If the heap is relatively empty, allocating new cells is fast, but as the heap becomes full, allocating new cells takes longer.

Conclusion Mark-sweep collection usually requires the maintenance of a free-list that needs to be updated at each collection and accessed at each allocation. At every garbage collection cycle, each live cell has to be checked and an associated mark-bit set. Then the collector has to visit every cell in the heap linearly and return dead cells to the free list. On top of this, classic mark-sweep algorithms tend to fragment memory with cells located at different places in the heap. This in turn has implications at two levels. First, the locality of objects is degraded since objects are scattered all over the heap, which can lead to performance degradation through poor cache behaviour. Second, the time required to find a suitable memory slot for each allocated object can be high since the allocator needs to find a slot big enough for that particular object but still small enough to avoid wasting too much space. However, if response-time is not of critical importance, this algorithm offers a better throughput than reference counting.

Because the algorithm visits every live cell during marking, and all dead cells during the sweeping phase, the complexity of mark-sweep is proportional to the size of the heap, not to the amount of live objects.

Copying Collector

This algorithm became popular after Cheney's discovery of a non-recursive implementation in 1970 [38], improving upon Fenichel and Yochelson's recursive

copy-collector [55]. Even today, copying collection is part of many state-of-the-art algorithms, and the basis upon which *generational collectors* build.

Unlike mark and sweep, copying collectors divide equally the heap into two *semi-spaces*. One of the semi-spaces is used to store current data (*Fromspace*) while the other semi-space contains free space (*Tospace*). Although copying collectors require twice as much memory to run the collector compared to a mark and sweep collector, Cheney's algorithm does not require any additional space to store meta-data or auxiliary data-structures. Copying collectors identify live objects by traversing the graph (in *Fromspace*), and recursively copy every live object into *Tospace*. Once all live cells have been copied to *Tospace*, *Fromspace* is considered to contain only garbage and the roles of *Fromspace* and *Tospace* are reversed for the next collection.

Cheney's copying collector In order to save CPU-time, space usage and avoid stack-overflows, Cheney implemented the first iterative copying-collector [38]. His implementation makes use of two pointers, *scan* and *free*, pointing to each end of a queue. When the GC starts, the algorithm flips the roles of *Tospace* and *Fromspace* and initialises the two pointers to point to the bottom of *Tospace*. Then, the roots are copied into *Tospace* and the cell pointed by *scan* is scanned for pointers that have not been copied yet. When live cells are found, they are copied into *Tospace* at the location pointed at by *free*. Every time an object is copied, a forwarding pointer is left in *Fromspace* so that unprocessed live cells pointing at this object can update their references. The reference of the original pointing cell is updated with the new location and *free* is moved along by the size of the object copied while *scan* is moved along by the size of the object scanned. Termination happens once the *scan* pointer catches up with the *free* pointer, meaning all live cells have been copied.

Conclusion The biggest attraction of copying collection comes from the fact that the amount of work to be performed is directly proportional to the number

of live objects. This makes the algorithm particularly attractive if the amount of live objects to be copied is small. Copying collectors also exhibit an important side-effect in that by copying data from one space to another, it naturally performs a compaction of the heap into fewer pages, diminishing the size of the program's working set. This may improve locality if related data is kept together [65, 139, 85, 140]. One of the biggest advantages of copying collectors is that they allow very fast allocation by using a *bump-pointer allocator*, where the next free cell is simply the following cell in the heap. This avoids the potentially expensive search for a suitable free memory slot, and object sizes are naturally handled.

Unfortunately, copying collectors have several drawbacks. First, the memory requirements are high since at any one time, half the heap, usually referred to as *copy reserve*, needs to be free to copy GC surviving objects into. Second, objects with a long lifetime are repeatedly copied from one semi-space to another during GC. Third, Cheney collectors perform a breadth-first search which will change the layout of the graph and may interfere with locality [170, 140].

Finally, while it is very efficient when few cells are live, the cost of copying objects can be particularly high when many objects are live at the time the GC is triggered.

Mark-Compact

One of the major drawback of mark-sweep collectors is their tendency to fragment the heap. This is especially true when a variety of objects of different sizes have to be handled. In a fragmented heap, after a GC, it may happen that there are many small 'holes' available in which new objects can be allocated. These, may not be big enough to accommodate a large object, even though the total amount of free space is sufficient. A standard mark-sweep collector would then have to expand the heap or abort execution. In terms of allocation, while mark-sweep allocators typically have to find a place to allocate a new object into, allocation in a copying collector can be done extremely fast using a bump-pointer.

A solution to solve these issues consists in *compacting* the heap. A compacted

heap consists of a contiguous area of live data at one end of the heap, and a contiguous area of free space at the other end. Compacting is usually performed in three passes over the heap. The first phase consists of marking live objects in the graph. The second phase and the third phase interchangeably consist of compacting the graph by relocating cells, and updating the values of pointers that refer to moved cells.

A study by Warren [167] shows that some objects remain at the bottom of the heap and never die throughout computation. He referred to these quasi-immortal objects as a *sediment.*

The open-source HotSpot Java Virtual Machine [113] uses a mark-compact algorithm to manage the old generation (see Section 2.1.3). HotSpot allows the size of this sediment to be varied through a parameter that controls the degree of fragmentation of the oldest prefix of its old generation: this prefix is not moved at compaction time.

By making the heap contiguous, a mark-compact collector can perform fast allocation like a copying collector. It improves over a copying collector in the fact that it does not require a copy reserve, and can be tuned to avoid repeatedly copying objects with a long lifetime [113]. Mark-compact may also improve the spatial locality of objects in the heap and reduce the number of page faults. Despite these advantages, mark-compact collectors are very expensive because of the number of phases necessary to compact data.

2.1.3 Generational GC

To the end-user, the biggest disadvantage of a traditional tracing collector is the often prohibitive pause-time introduced during a GC. To a large extent, this can be attributed to the need for scanning large heap regions.

Another weakness of these algorithms is their inability to take advantage of the fact that every object is unique, has its own purpose, its own lifetime, its own size, its own type, its own connectivity with other objects, and the method that allocated it has its own calling context etc.. Many studies have been published

on the lifetime of objects, and there is clear evidence that the lifetime of objects is not uniform, nor is it random [52, 57, 163, 172, 74, 4, 169, 131, 134]. However, traditional tracing collectors such as mark-sweep collectors and copying collectors treat all objects uniformly.

By partitioning the heap into separate smaller *regions* (or *partitions*, or *spaces*), objects can be segregated according to various attributes and collect these partitions independently [160, 161, 162, 155, 159, 156, 41, 79, 66]. To reduce pause-times, regions that are likely to contain the most number of dead objects might be garbage collected first, and if not enough space can be reclaimed, then other regions can be collected.

Collecting small regions with mostly dead objects can drastically reduce pause-times.

Partitioning The Heap Into Regions

Bishop suggests a practical way of reducing GC pauses (usually induced by traditional tracing collectors) which consists of splitting the heap into two or more regions [19]. Allocation into each of these regions is performed according to predefined policies.The most common criteria based on which objects are allocated together are explored below. These include the age the object to be allocated, the size of the object, the connectivity of the object towards other objects, and the class of the object [85].

The heap organisation used in this scheme allows for regions to be garbage collected independently, or even reclaimed as a whole [160, 161, 162, 155, 159, 156].

The effectiveness of this scheme relies on several factors. First, it allows for smaller portions of memory to be garbage collected, rather than the whole heap. Second, by targeting regions where most objects are likely to be dead, reclaiming unused space can be fast. Third, splitting the heap into regions offers the possibility to choose the most appropriate GC algorithm for each region, based on known characteristics of the objects within it. For example, a region which contains mostly large objects would probably be best suited for a non-moving

garbage collection algorithm.

Since objects are allocated into different regions, some objects may point to objects in other regions, therefore creating *Inter-Region* pointers. This implies that a region should not be collected independently without considering incoming pointers as extra GC roots.

Tracking inter-region pointers is performed by using a *write-barrier* (write-barrier techniques are reviewed later in this section). This usually consists of a small piece of code emitted by the compiler, but some implementations for specific architectures implement a hardware trap [90, 77]. It is in charge of monitoring all writes of pointers, and adding every object with a cross-region pointer into a specific data-structure (see later in this section). This data-structure is then used as an extra root-set when performing a region-independent GC. Another drawback is that region-based GC becomes inefficient if most objects within a region are not dead at the time of collection.

The paragraph below reviews different policies used to group objects within regions.

Tofte And Talpin's Scheme Tofte and Talpin provide region-based memory management [160, 161, 162, 155, 159, 156]. They store all values in a stack of regions at run-time, and each of these regions is lexically scoped. As a consequence, regions do not need to be garbage collected, but are instead reclaimed as whole once the reference to a region is no longer available in the stack. They prove that stack allocation is safe, which entails that no deallocated cells are used by the rest of the program. Regions can grow dynamically and store recursive types, such as lists and trees.

However, their analysis is specific to functional programming.

Region Analysis and Transformation for Java Programs Building upon Tofte and Talpin's idea, Cherem and Rugina [41] propose a static analysis that automatically transforms standard Java code to add region specific code. Their

analysis is capable of handling the imperative, object-oriented constructs of the Java language, whereas Tofte and Talpin's approach was used in the context of a simply typed lambda calculus functional language.

Their analysis works on any Java program and does not require any help from the programmer. They find that a large fraction of objects can be placed in regions or on stack, but not all.

Their system yields significant absolute memory savings for several benchmarks against a non-GCed system. However, it sometimes shows absolute memory increase compared to a garbage collected system. Another drawback of this technique is the need for an offline static analysis of the Java program prior to running it, which in their case accounts for around 16% of compilation time. Unfortunately, they do not perform performance analysis, and the benchmarks they use are small (Java Olden [34]).

Large Object Space Large objects can be considered a special group of objects because they are more costly to allocate and to copy. Because of this, many garbage collectors allocate large objects in a separate region managed by a non-moving GC. A number of implementations incorporate this strategy [36, 86, 130, 165, 21], some not collecting it at all, some managing this Large Object Space (LOS) with a mark-sweep collector, and some using a treadmill GC (an incremental non-moving garbage collector [11]). Hicks et al. survey the different techniques used for the management of large objects spaces and identify a wide variety of issues that might affect its design [76].

Immortal Space Objects that are used throughout computations and are only deallocated at the very end of the program are called *immortal* objects. If one knows ahead of time which objects are going to be immortal, or at least are very likely to be, it makes sense to handle these objects specially. Several GC implementations now use an immortal space where all objects known or believed to be immortal are allocated [22, 21, 135, 30, 144, 24]. Objects within the immortal

space are added to the GC roots and the immortal space is never garbage collected.

Finding Your Chronies On the basis that connected objects share similar lifetimes [79], Guyer and McKinley [66] seek to colocate them in the same space. They combine an inter-procedural static analysis, that identifies the object to which a new object might be connected, with a specialised allocator, that places the new object in the same space as the object to which it is connected. By doing a basic collocation analysis, they build a graph that represents a conservative approximation of the connectivity between objects.

As well as reducing copying, colocation also reduces pressure on the write-barrier (see the following section). They tune their algorithm to prevent overly aggressive colocation from increasing pressure on the mature-space. Experiments with jvm98 show that GC time can be reduced by up to 75% [66].

Write-barrier

Region-based GC, objects are segregated into different regions. Ideally, the algorithm should garbage collect the region with the most objects likely to be dead.

However, care has to be taken when designing mechanisms to garbage collect regions independently. When collecting a region independently, other regions are not traced, and a mechanism to discover which objects within the region to be collected are being referenced by objects in other regions is therefore needed. Objects within the region to be collected that are referenced by objects in other regions should therefore be considered live.

To overcome this problem, researchers usually record references of objects pointing to other regions that have to be collected independently. This record is then considered part of the GC roots, and it can be implemented in several different ways. When performing a region-independent collection, it is then possible to identify every object that is being referenced by objects within other regions. More generally, if one wishes to collect a region B independently from a region A (or simply collect region B before region A), it is necessary to keep a record of

pointers from region A to region B.

Unfortunately, the use of such records raises another concern. During the execution of the program, references between objects are modified. It is therefore necessary to ensure that the records are up to date. This is done using a *write-barrier* which tracks, during program execution, all writes of pointers from one region into another region that need to be collected independently. When an object creates a pointer to another object located within a region that is collected independently from the others, the records need to be amended to reflect the change. This process of updating records is usually referred to as *write-barrier slow path*. On the contrary, the *write-barrier fast path* is used when a pointer write has been trapped but does not need to be recorded (see below).

Write barriers are expensive because of the extra computation required to check every time a pointer write occurs. The price to pay to record a pointer from one region to a region that needs to be collected independently is even higher.

Recording Inter-Region Pointers

A write barrier needs an associated data-structure to keep track of the pointers from an old to a younger generation. This data-structure can be implemented in different ways the most popular techniques are reviewed below.

Entry tables Entry tables were used by Lieberman and Hewitt [106] as part of the first generational collector. They associate an entry table with each region that requires independent GC. During the execution of the program, if an object X is to point to an object Y in a region that needs to be collected independently, an entry pointing to object Y is created in the table associated with the region of object Y, and object X is updated to point to the appropriate table slot. If object X is already pointing to an entry in the table, the reference is updated to the new entry, and the old entry is destroyed. The advantage of this algorithm is that in order to collect independently a region, it is not necessary to trace every other region. Instead, it is only necessary to scan the entry table associated

with the region to be collected. However, if two objects refer to the same object in a region that needs to be collected independently, multiple references to the same object will be added to the entry table. Also, the costs associated with this technique are acceptable on a specifically designed machine, but would be very high on commodity hardware.

Remembered Sets Instead of keeping a table of entries to objects referenced by objects in other regions, Ungar [163] records objects pointing to a region requiring independent GC using a *remembered set.* At each point in time, if an object points to another object which is located in a region that needs to be garbage collected independently, the location of the parent object is added to a remembered set. Also a dedicated bit is added to the header of each object to avoid duplicates in the remembered set. Under this policy, scanning time is dependent on the number of objects remembered. Drawbacks include the fact that checks are repeated if an object is stored into several times between two GCs. Also, the cost of scanning objects pointed to by the remembered set at collection time can be high if objects pointed to are large. Despite these drawbacks, remembered sets remain a popular choice today.

Sequential Store Buffers Hudson and Diwan fill a fixed-size *Sequential Store Buffer* (SSB) with addresses that might contain pointers to regions that need to be collected independently. These addresses are added at the end of the SSB and buffer overflows are trapped using a 'no access' guard page. Adding addresses to the SSB is very quick, and can even be done in just two instructions if the pointer to the next free slot in the buffer if kept in a register. Values are moved to the remembered sets on two occasions: if the SSB overflows, or at collection-time. By building the remembered sets as circular hash tables using linear hashing, the algorithm can ensure that no duplicate addresses are added to the remembered sets [81].

Card Marking Another popular alternative is *card marking* [143, 82, 6]. The heap is divided into many small regions (cards). The size of these regions can be made smaller or larger to optimise locality of reference or reduce scanning time but are generally less than the size of a page. A table of bytes is associated with these cards where each entry in the table corresponds to one card in memory. Every time the pointer location of an object is modified, the byte corresponding to the card in which the object is located is set (*dirtied*). A second-level data table called the *segment modification cache* is usually used, where each entry corresponds to an entire table of card-tables [143]. Every time a card is dirtied, the associated bits in the tables at both levels are dirtied. At collection-time, the memory regions covered by each dirty card need to be scanned for inter-region pointers. The memory requirements for this technique are low: less than 1 percent of the heap for a 128-byte card. On the other hand, this technique is less precise than remembered sets which records at pointer-level.

Generations

In 1976, Deutsch and Bobrow identified that newly allocated objects are likely to be abandoned within a relatively short time [52]. In 1981, Foderaro and Fateman found that at each GC, over 98 percent of dead objects to be recycled have been allocated since the previous garbage collection [57]. In 1984, Ungar introduced the famous *weak generational hypothesis* which identifies that most "objects die young" [163]. Since then, several studies have confirmed Ungar's findings. In 1989, Zorn identified that between 50 and 90 percent of Common Lisp objects die before they are 10 kilobytes old [172]. In 1991, Hayes observed that in the Cedar language, a Modula-like language developed at Xerox, only 1 percent of objects survive beyond 721 kilobytes [74]. In 1992, Appel observed that over 98 percent of any given generation is reclaimed at each collection [4]. In 1994, Wilson observed that typically 80 to 98 percent of objects die before one further megabyte of heap storage has been allocated [169]. In 2006, Jones et al. analysed the demographics of objects, arguing that garbage collectors should be lifetime

aware [92]. These studies provide strong evidence that a vast majority of objects have a short lifetime, while a small proportion of objects may live a very long time.

Generational garbage collection was developed to take advantage of these observations by focusing on newly allocated objects. In this garbage collector, the heap is divided into two or more regions called *generations*. The number of generations varies between implementations. For instance, an early implementation of Standard ML of New Jersey used two generations, when Tektronix 4406 Smalltalk used seven [2, 36]. Hudson et al.'s University of Massachusetts Language-Independent Toolkit [86] was capable of dynamically varying the number of generations. To facilitate the description of this algorithm, it is assumed that a generational collector has only two generations, but observations apply equally to a generational garbage collector with more than two generations.

In a generational collector, new objects are typically allocated in a region usually referred to as *young generation* or *nursery*. When it is no longer possible to allocate new objects due to lack of available heap space, a garbage collection of the nursery is triggered. This is called a *minor collection*. After a minor garbage collection, surviving objects are usually copied (promoted) into the old generation which is collected less frequently. However, if not enough space can be reclaimed by doing a minor garbage collection, then a *full* (or *major*) garbage collection is triggered, and both the young generation and the *old generation* (the *mature space*) are garbage collected. Since young objects are allocated in the nursery and older objects promoted to the mature space, objects in a generational GC are in essence *segregated* by age. Because the young generation can be collected independently, the use of a write barrier and a structure to hold references to pointers from the old generation to the young generation is necessary (see below). For simplicity, this section assumes that this structure is a remembered set.

The youngest generation is usually comparatively small and is collected much more frequently than the old generation, because it is expected that most objects in the young generation will be dead. Write barriers and remembered sets in the

context of a generational collector are reviewed below, followed by the description of some interesting garbage collection techniques taking advantage of object lifetimes. At the end of this section, the efficiency of generational garbage collection is reviewed.

Write Barrier And Remembered Sets For Generational GC

Generational GCs split the heap into different generations (regions), and the young generation is always collected before the old generation. Therefore, generational GCs require a write-barrier and a unidirectional remembered set (not bidirectional) to hold references to pointers from the old generation to the young generation. Having a unidirectional remembered set reduces dramatically the number of pointers that need to be remembered, since most pointers are from young objects to old objects. Most writes are to young objects located in the nursery, and many of these objects are close to each other [148].

Pointers from the old to the young generation are relatively rare because they only arise through (usually destructive) assignments such as "putfield" and "putstatic" in Java. Therefore, write-barriers in Java need only check the "putfield" and "putstatic" assignements. When an old-to-young reference is created, the write-barrier needs to store the old object in the remembered set. This process is called the *write-barrier slow path*. On the contrary, the *write-barrier fast path* is used when a pointer write has been trapped but does not need to be recorded. For example, in the case of an old-to-old, young-to-young, or young-to-old pointer write.

Appel-Style Collector

In 1989, Appel introduced an elegant, efficient and easy to implement GC with a fast allocation time [3]. This collector, often known as the *Appel-style collector* comprises of only two generations. The nursery size is flexible, giving objects longer to die, hence reducing the chance that a young object will ever be copied. In his implementation for the Standard ML of New Jersey compiler, only 2% of

objects are expected to survive a minor GC.

After a major GC, the unused portion of the heap is divided into two equal-sized parts. The first part acts as a copy reserve where survivors of a minor collection will be copied into, and the second part acts as the free region. When the remaining free space is exhausted, a minor GC is triggered. He estimates that a minor GC is about fifty times faster than a major GC. After a minor GC, if the old generation occupies around half the heap, a major GC is triggered.

This algorithm is still popular today, and was used by Blackburn et al. in their pretenuring work, upon which this thesis builds (see Section 2.4.7). In order to compare this work to theirs, an Appel-style garbage collector is also used in the experiments.

Age-Oriented Garbage Collection

Paz et al. propose a garbage collector similar to a generational GC in the sense that it segregates objects by age [123]. They implement their collector for a multi-processor environment where throughput is a concern, but pauses times are not. They argue that by using an *on-the-fly collector*, which does not need to stop the program threads simultaneously to collect garbage [54, 122], these pause times are naturally very short. Unlike a traditional generational collector, they always perform a full heap collection, meaning they do not have to be concerned about remembering old-to-young generation pointers. Old objects are managed using a reference counting GC, and young objects using a mark-sweep collector. The interest in this scheme with regards to this thesis is limited to showing the reader the possibilities offered by lifetime-aware collectors, and the reader is therefore referred to Paz et al.'s paper for more information on this complex system.

There are a few advantages to this technique such as the fact that they do not need to record inter-generational pointers. Also, in a multi-threaded environment with several processors, their implementation has the advantage or requiring less collections than a typical generational collector, therefore requiring less thread synchronisation and reducing the time spent on GC initialisation. Of course, the

downside of this approach is that pause-times are increased since they always collect the full heap.

This approach is interesting to this thesis as it shows a different way to make use of object lifetimes. This thesis focuses on predicting the lifetime of objects, which can then be exploited by various lifetime aware garbage collectors, such as this scheme and the following ones reviewed.

Older-First Garbage Collection

Generational collection helps improve the efficiency of garbage collection in fast-allocating programs by focusing its garbage collection efforts on young garbage. However, pauses introduced by full heap collections, when they occur, are still comparatively long. Stefanović argues that current generational collectors focus on the fact that "most objects die young" but he states that the very newest objects are patently not garbage [148]. Therefore, a technique known as *older-first* garbage collection was introduced to give objects more time to die [71, 72, 148, 149].

Older-first (OF) garbage collectors avoid collecting the very youngest objects that have not had enough time to die. The heap is divided into an allocation region A and the copy region C. New objects are allocated at the back of the region A. Each region contains several *windows*, and whenever the heap is full, the window located at the front of region A (the older objects) is collected, and surviving objects are copied into the back of region C. When region A becomes empty, the roles of region A and region C are flipped.

This collector requires the write barrier to remember more pointers than a generational GC, because all pointers from younger to older windows need to be remembered. When a window is collected, surviving objects with pointers to older windows are added to the remembered set.

Interesting findings include the fact that the number of recorded younger-to-older pointers is typically between one and three times the number of recorded older-to-younger pointers. In terms of performance, older-first collectors are particularly effective at managing data with queue-like or random lifetimes and heaps

containing large amounts of live data. However, as one would expect, their performance dramatically decreases if the scavenger allocates mostly short-lived objects.

Beltway: Getting Around Garbage Collection Gridlock

Blackburn et al. generalise the concept of generational garbage collection by proposing the Beltway framework [23]. This framework combines and exploits five important principles that are important to copying GCs:

1. The weak generational hypothesis that "most objects die young".
2. Avoid collecting old objects (this is what generational GCs do).
3. Use incrementally to reduce pause-times.
4. Use of small nurseries to improve locality.
5. Give the very youngest objects time to die, like in the older-first GC.

This framework segregates objects by age incrementally. Objects are grouped into segments of memory called *increments*, and increments are grouped into regions called *belts*. The number of increments and belts can be configured using command-line options.

Using these options, their collector can be configured as a semi-space, generational, or older-first collector. The command-line options allows them to specify any number of belts and any number of increments. New objects are allocated on the last increment of the lowest belt (nursery belt). Objects surviving a GC are promoted into the next higher belt, if there is one. When a GC occurs, the oldest increment on the lowest belt is collected first.

The beltway framework is very efficient at exploiting the lifetime of objects. Objects are segregated by age, and are given time to die. It is more flexible than the older-first GC, because the size of each increment and the number of belts can be tuned. With regards to this thesis, an oracle capable of predicting the object lifetimes could allocate objects more efficiently into the most appropriate increment.

The Beltway framework allows the easy experimentation of different garbage collection configurations. Their collector is efficient, and they are able to demonstrate important throughput performance gains over other generational collectors by up to 40%.

Efficiency

Generational collectors use a heuristic (the weak generational hypothesis) as a basis and have proved very successful on many occasions. They are often regarded as the best choice for non real-time implementations. During minor collections, pause-times can be reduced drastically for some applications. Paging is often improved since the collector usually deals with a fraction of the memory at each collection. Each generation of a generational GC can be managed by its own garbage collector algorithm, giving more flexibility to the GC implementer and offering more scope for optimisation. Also, the nursery can be collected independently, reducing the average pause time (not worse case).

On the other hand, recording pointers from old generations to young generations requires an extra structure, which takes space, and requires a write-barrier which is expensive in terms of computation time. To reduce overheads, *only* pointers from the old generation to the new generation are tracked, which implies that the mature space cannot be collected independently. This issue could be tackled by also recording pointers from the younger generation to the older, but the costs associated with tracking young-to-old generation pointers would be prohibitively high because they are much more common than old-young pointers. Another concern of generational collectors is that they require a copy reserve where surviving objects are copied into. For the Appel-style collector, the space overhead is 100% since half the heap needs to be available as copy reserve at any point in time.

Although exhibiting a number of enhancements over other collectors, generational collectors have a core problem: while most programs allocate objects which tend to die young, some programs do not and this can defeat the weak generational hypothesis. Such behaviour would not only make minor collections inefficient, but

will also tend to tenure many objects in the older generations, hence reducing the space available for the nursery (in the case of an Appel-style collector [3]), and requiring more frequent expensive major collections. Baker emphasises this problem and shows an example which defeats the weak generational hypothesis [12] based on the physics *Radioactive Decay Model*. He then generalises to garbage collection to show that in some cases, the weak generational hypothesis does not apply.

2.2 Object Lifetime Prediction

Today, even leading edge garbage collectors can not predict the future. They are incapable of knowing ahead of time for certain how long an object is going to live.

If they were able to, garbage collection could be made extremely cheap, and more efficient object allocation could be performed in the heap. This could be achieved by allocating objects of similar lifetimes together in the heap into regions to which a time of death would be associated ahead of time. A region could then be collected as a whole, once its time of death had passed.

Over the past twenty years, a number of studies have tried to predict object lifetimes. Some research has focused on predicting object lifetimes using pointer-analysis, a technique that establishes which pointers, or heap references, can point to which storage locations [133, 51]. However, pointer-analysis is expensive to compute accurately, even more so for complex programs or systems with dynamically loaded libraries. This thesis does not investigate pointer-analysis literature because it is not directly relevant to the work presented, but the reader is invited to refer to Hind's paper for an interesting discussion on the topic [78].

This section presents the most relevant studies on object lifetime prediction with regards to this thesis.

2.2.1 Using Lifetime Predictors to Improve Memory Allocation Performance

Barrett and Zorn propose a profile-based system capable of accurately predicting objects that will be short-lived at allocation time [14]. Their predictions are based on the size of the object at the time of allocation, as well as the call-chain: an abstraction of the program's call-stack at the time that the event occurred. Lifetime data is gathered from instrumented C programs which maintain the current call-chain state and associate each object with its call-chain. At each allocation, the object and its call-chain are added to a database. Their data shows that over half of the objects in each program live less than 10,000 bytes. They defined an object to be short-lived if it lived less than 32 kilobytes and found that short-lived objects accounted for more than 90% of all bytes allocated in every program.

Once lifetime data has been gathered, they reuse the lifetime database to drive the allocation of short-lived objects by performing a non-instrumented run of the same program. At each allocation, they determine if an object is certain to be short-lived by checking its size and call chain against the database of lifetimes. Short-lived objects are allocated into dedicated areas, in order to make allocation and deallocation faster. When an object within a dedicated area is *freed*[3], a counter recording the number of objects in the area is decreased. When the counter drops to zero, the entire area is reclaimed.

Their paper demonstrates that a call-chain of events combined with the size of an object can efficiently predict object lifetimes. However, while this offers great precision, less predictions can be made and checking the call-chain at each allocation is expensive.

[3]Remember this is a C program, where allocated memory must explicitely be freed

2.2.2 Predicting Lifetimes In Dynamically Allocated Memory

Cohn and Singh extend Barrett and Zorn's work by data-mining the lifetime database [46]. They create decision trees (see Section 2.5) based on the top 20 words in the stack and the size of the object allocated. From the trees, they derive rules that predict short-lived objects and immortal objects (which they call *permanent*) from other objects. However, using Barrett and Zorn's memory organisation, allocating an immortal object into a memory block for short-lived objects may make the entire block non-reclaimable.

The results show interesting potential in applying machine learning techniques to object lifetimes predictions. Despite being a short paper, this study is interesting for this thesis, as new ways of predicting object lifetimes using data-mining are explored.

2.2.3 Segregating Heap Objects by Reference Behavior and Lifetime

Using profile-based information, Zorn and Seidl show that the behaviour of objects in the heap is predictable [171] and can be used to reduce page faults. In previous work, they identified that for some programs, a small number of objects receive a majority of the references, while other objects receive almost no references [138]. Building upon this work, they suggest dividing the heap into four different segments which are: *highly referenced* (HR), *not highly referenced* (NHR), *short-lived* (SL) and *other*.

To predict object behaviour and reference density, they analyse program features such as:

- *Stack pointer*: value of the stack pointer at the time of heap allocation.
- *Path point*: one specific point in the call-chain of events.
- *Stack contents*: subset of the call chain at the time of allocation.

- *Object size.*

Based on measurements taken from six allocation-intensive programs written in C, they find correlation between object characteristics and the four different segments. They use program features as predictors of object lifetimes, and perform a cross-validation to avoid the case where a program could predict itself. In other words, each program is used to predict a different program, but never itself. To perform experiments, they use a wrapper around the allocation method to determine what segment to place the objects in based on the predictors described.

Their scheme allows for a significant decrease in page faults (up to 95% in some memory sizes), and confirms that the behaviour of objects can be predicted and exploited.

2.2.4 Object Lifetime Prediction in Java

A recent study by Inoue et al. also highlights the relation between the state of the stack and the allocated object's lifetime [89]. They predict lifetimes using information such as the call chain and the type of the object.

To gather relevant lifetime information, they collect program traces recording the times of birth and death of each object, the object identifier, its type, and the call chain. They identify that a large percentage of objects (at least 13% in for the jvm98 benchmarks) die as soon as they are allocated, and suggest that an optimising compiler might take advantage of this discovery. Surprisingly, they show that in some applications, it is possible to predict the lifetime of an object to the byte.

Unfortunately, they do not report performance tests as they do not possess a virtual machine tuned with specialised allocators.

This section has shown that predicting the lifetime of objects is possible, and that there exist ways to take advantage of such prediction. Section 2.3 reviews object lifetime predictions in the context of a generational collector where objects with a long lifetime are allocated directly into the mature space.

2.3 Garbage Collection Discussion

2.3.1 Summary

In the previous sections, three main families of garbage collection algorithm were examined.

The first one is reference counting, which by keeping a counter of the number of references to any object at any time aims at reclaiming objects as soon as they become garbage. This algorithm is interesting because garbage collection is interleaved with mutator activity, allowing for very small pause-times. At every pointer write, reference counts of objects are updated, and dead objects are reclaimed. However, it was also shown that this algorithm has weaknesses such as the inability to naturally reclaim cyclic data-structures. Reclaiming cyclic data-structure requires the use of an extra tracing collector [168], or an additional data-structure to store pointers to cells with multiple references, which are potentially part of a cyclic data-structure [107]. Furthermore, while the basic reference counting algorithm aims at reclaiming objects as soon as they become garbage, doing so is too costly because of the maintenance imposed by keeping object reference counts up to date. Often, deferred reference counting techniques are used to reduce the processing costs of keeping references up to date, but by doing so, the algorithm is no longer able to reclaim objects as soon as they die [52, 24]. Finally, the space requirement of a reference counting algorithm is high since it requires extra room in each object's header to keep the reference count.

The second family of algorithms discussed are tracing collectors. They are able to reclaim cyclic-structures naturally. There are two main sorts of tracing collector, which are mark-sweep collector, and copying collectors.

Mark-sweep has a lower overhead than reference counting, but still requires the storage of the mark-bit in the header of each object. It requires the maintenance of a free-list that has to be updated at each collection. Mark-sweep algorithms have fragmentation issues, and the amount of work to be performed at each GC is proportional to the size of the heap. Also allocation can be slow due to the need

to identify a memory slot suitable for the object to be allocated in. However, its throughput is higher than reference counting.

Copying collectors perform allocation in a linear manner, using a bump-pointer. Allocation is very cheap because new objects can be allocated in the next free cell, which is simply the following cell in the heap. The cost of copying GCs is proportional to the amount of live objects that have to be copied. Therefore, a very large heap with few live objects can be reclaimed cheaply. Copying GCs perform a compaction of the heap naturally at every GC, which has positive repercussions for paging. Finally, copying GCs can perform very fast allocation. The main drawback of this algorithm is the fact that only half the heap can be used at any one time.

The third family of GCs explored are generational GCs.

The disadvantage of generational collectors lies in the fact that in order to garbage collect the nursery independently, the algorithm needs to be aware of all objects within the nursery that are referenced by objects in an older generation. This is performed using a write barrier that tracks pointer writes from older generations to the nursery and records such pointers in a data-structure such as a remembered set. In this algorithm, the time overhead of the write-barrier and the space overhead introduced by recording old-to-young pointers are the main concerns. Another concern is the fact that this algorithm is based on the assumption that most objects dies young. This is true for many programs, but not all, and for mutators that do not exhibit this hypothesis, a generational GC would not perform well.

By allocating young objects together, this algorithm can perform nursery GCs cheaply because few objects typically survive a GC and need to be copied into an older generation. In most cases, generational GCs perform well, and they are the basis of many high-performance garbage collection algorithms today.

2.3.2 A Unified Theory of Garbage Collection

Having reviewed the major garbage collection algorithm, Bacon et al.'s *unified theory of garbage collection* is now reviewed [9]. These authors observe that while reference counting algorithms and tracing algorithms have universally been viewed as being fundamentally different approaches, the more they are optimised, the more similar they become. These authors in fact see these two types of garbage collectors as dual to each other, much like "matter" and "anti-matter". Reference counting collectors concentrate on dead objects (anti-matter), and traverse the graph of dead objects starting from other dead objects with a reference count of 0. Tracing collectors concentrate on live objects (matter), and traverse the graph of reachable objects, starting from live objects (GC roots). They show that high-performance collectors such as deferred reference counting and generational collectors are hybrids of tracing and reference counting collectors. Mark-bits (forwarding addresses) are seen as playing similar roles to reference counts. E.g. mark bits (0 or 1) and reference counts are approximations – because of cycles – of the number of live references to any object.

The most fundamental advantage of reference counting is the fact that it is incremental by nature. But because the run-time overhead of a basic reference counting implementation is much too high, deferred reference counting techniques are often used [52, 24], in which references from stack frames are accounted for separately. Doing so, however, results in delayed reclamation of garbage and longer pause-times, just like a tracing collector.

Conversely, high-performance tracing collectors reduce the long pause-times imposed by this algorithm by implementing generational garbage collection. By doing so, they introduce the need for a write-barrier and a structure to store old-to-young pointers, therefore introducing a regular run-time overhead, much like in a reference counting algorithm.

Bacon et al. then compare algorithms against each other and show that tracing collectors and reference counting algorithms are really opposite to each other.

As optimisations are applied to each of these algorithms, they start taking characteristics of each other, and become similar in many ways.

This paper is interesting as it provides a higher-level view of garbage collection algorithms, and gives a better understanding of the trade-offs involved in designing high-performance GCs.

2.4 Pretenuring

2.4.1 Concept

In a generational GC, young objects are allocated into the nursery, and at each collection, survivors are promoted/copied from the nursery to an older generation [3]. The efficiency of a generational GC relies on the fact that most objects die young. While this is often true, inefficiencies arise when objects have a longer lifetime than expected [12, 26, 22].

By knowing ahead of time how long an object is going to live, objects can be allocated in a more efficient manner. For instance, objects with a short lifetime can be allocated in the nursery, while objects with a longer lifetime can be allocated directly in the mature space. This technique, referred to as *pretenuring* can save the unnecessary copying of objects from young to old generations. By knowing all the objects allocated in the nursery are dead at the time of the collection, a young-generation collection is virtually free since the whole region can be reclaimed as a whole. In this ideal world, a write-barrier, and its associated run-time overhead, would not be necessary since it is known that all the objects in the young generation are dead. Furthermore, to avoid processing immortal objects several times, these can be allocated into a separate "immortal" space which will never be collected.

A popular unit to measure the efficiency of a garbage collector is the "mark/-cons" ratio. This is the number of bytes that the collector copies divided by the number of bytes it allocates. This allows researchers to measure the amount of

work done by a copying collector. Higher ratios mean the collector will require more time, because more objects must be copied. The goal of any pretenuring system is to reduce the mark/cons ratio as much as possible, by reducing the amount of copying required.

In a pretenuring scheme, it is important to keep a conservative approach when trying to predict object lifetimes because pretenuring objects wrongly can have a negative impact on performance. In the case of an Appel-style collector [3], it would reduce the space available for the nursery since the nursery size is flexible and is equal to the size of the heap minus the size of the mature space (and sometimes an immortal space). This would also lead to more frequent nursery collections (because the nursery size is reduced), and more frequent full heap collections if there is too much garbage in the mature space.

Furthermore, it may result in some young-lived objects, which would otherwise have been dead, to be promoted after a minor collection only because they are referenced by a wrongly pretenured object. This effect is known as *nepotism* [164]. Allocating objects wrongly in the immortal region can have a larger negative impact on performance since none of their referents would ever be collected, and would have to be processed at every major GC.

The potential offered by a well-designed pretenuring system is high, but even then, pretenuring has drawbacks. For instance, because some objects are allocated in the nursery, and some in the mature space, the locality of objects may be disturbed with respect to a non-pretenuring scheme, resulting in damaged cache performance at run-time. Also, by allocating some objects directly into the mature space, pretenuring may increase the number of old-to-young pointers that have to be trapped by a write-barrier. This not only increases the number of pointers that have to be processed by the GC, but also increases the run-time overhead of the write-barrier which may have more pointers to add to the remembered set (or any other suitable structure).

2.4.2 Self-Prediction and True Prediction

Predictions based on the analysis of a previous program recording are called *program-specific* predictions, or *self-prediction*. For instance, Harris [73] records allocation sites that consistently allocate long-lived objects using an instrumented VM, and later reuses this information to perform pretenuring. This is self-prediction. This is also the case for CHL [84] and Jump et al. [96] because they first sample the lifetime of objects based on specific criteria, respectively the type of the object and the allocation site during the execution of a program. Once they have some confidence that a certain criteria is associated with long object lifetime, they can pretenure objects exhibiting this criteria.

The notion of self-prediction is opposed to the notion of *true-prediction*, whereby the lifetime of an object allocated in a program never encountered before can be predicted based on the characteristics of this object, such as its size, calling context, connectivity towards other objects, type, and the allocation site that allocated it.

In an abstracted view, self-prediction consists of predicting the behaviour of an item based on the past behaviour of that item, whereas true prediction consists of predicting the behaviour of an item based on the behaviour of other items.

2.4.3 Generational Stack Collection And Profile-Driven Pretenuring

In [39], Cheng, Harper and Lee (CHL) observe that savings might be made if long-lived objects are allocated directly into an older generation. Their implementation comprises a two-generation garbage collector where all new objects are allocated into a nursery and objects surviving a GC are copied into the old generation. Objects suitable for pretenuring are identified via program profiling, dynamic sampling at run time, or program analysis.

Each point in the program at which some memory allocation is performed is called an *allocation site*. In Java for instance, every *new* instruction in the

code is an allocation site. Dynamic sampling is performed by tagging each object with the allocation site in the program that allocated it, and inspecting tags of dead objects after each collection. From this profile, they identify those sites that allocate promoted objects consistently in their collector. An allocation site is said to consistently promote objects based on the fraction of objects that that get promoted out of all the objects it allocates. They found that over 96% data that is copied is allocated at allocation sites whose survival rate is at least 80%. In light of this finding, they use a threshold of 80% in their experiments, so allocation sites allocating more than 80% of promoted objects are considered for pretenuring.

Objects allocated from a site considered for pretenuring are allocated directly into the mature space. With this scheme, CHL observe savings in GC times of between 12% and 50%. While the speed improvements in terms of GC time are very good, dynamic sampling schemes usually come with run-time overheads due to the extra work that has to be performed throughout computation. In their experiments, CHL observe that a profiled program typically runs 50% to 200% slower than its unprofiled version, which is too expensive for any end-user. However, they are able to record these statistics and reuse them directly in a second run of the program where dynamic sampling is turned off. In this case, the execution time is reduced by 4% on average against the unprofiled version of their GC.

CHL's sampling scheme is very dependent on their specific configuration. For instance, they consider an allocation site for pretenuring if it consistently allocates objects that survive a GC of a specific configuration.This policy gives them GC time speedups, but a different setup triggering GCs at different times is likely to give different results. Therefore, this approach does not scale well to different GC configurations, different promotion policies or different heap sizes.

2.4.4 Dynamic Adaptive Pretenuring

In [73], Harris instrumented a JVM to sample object lifetimes at run-time and generate pretenuring decisions, therefore avoiding the need for a profile-gathering

phase. This paper is the first to consider profile-driven allocation in the context of an OO language with automatic memory management. His implementation comprises a two-generation generation garbage collector and an object sampling module.

A Local Allocation Buffer (LAB) is a thread-specific space in memory, usually allocated in the nursery, where a thread can allocate objects. LABs allow threads to allocate objects without having to worry about thread synchronisation as each thread has its own LAB. Therefore, most objects are allocated in the nursery using a bump-pointer within local allocation buffers.

To perform sampling, Harris uses *weak references*, which unlike normal references, are not followed by the garbage collector. This means that during GC, objects pointed by weak references will not be kept alive if all other references to these objects have been deleted. His object sampling is performed every time a thread-specific LAB overflows. An auxiliary data structure separate from the garbage collected heap records the relevant allocation site of each object being sampled along with a weak-reference pointing to this object. When a majority of sampled objects at a given site are long-lived, pretenuring decisions are made to ensure that any future objects allocated by that site would be allocated in the mature space. Reciprocally, using this *threshold-based* approach, if a majority of sampled objects at a given site are short-lived, then pretenuring decisions can be reversed. To allocate objects in the appropriate generation, Harris patches the code generated by the just-in-time compiler.

By reversing and re-enabling pretenuring decisions if they become harmful, this scheme can adapt efficiently to phased behaviour.

One inconvenience of the mechanism is that his work is specific to Java since it uses weak pointers.

A second shortcoming of Harris's approach is that object sampling is implemented as part of the LAB overflow handler, meaning objects will only be sampled when the buffer overflows, skewing samplings towards large objects. Instead, Jump et al. sample objects every n bytes of allocation [96], but this scheme also

skews sampling (see Section 2.4.6). Sampling has the problem of being reactive, which implies that pretenuring decisions can only be made in hindsight, which may be too late. The work presented in this thesis does not suffer this drawback.

As part of future work, Harris proposed to use objects types as predictors. This idea is implemented by Huang et al. where their sampling technique consists on sampling objects at every GC [84] (see Section 2.4.5) rather than on LAB overflow. Another alternative for pretenuring proposed by Harris consists of considering a combination of the class being instantiated, a single frame of allocation context and the allocation site. This would give much more precision to the analysis, but the cost of storing and computing this information is higher. To further increase precision, one could consider using the *call-chain*, which defines the sequence of functions of a program have been called up to a given point.

2.4.5 Adaptive Pretenuring Schemes For Generational GC

As explained earlier in this chapter, objects may have many different attributes such as lifetime, size, type, connectivity with other objects, its own calling context etc.. Huang et al. propose a dynamic sampling mechanism to evaluate class-type based predictors, as opposed to allocation site based predictors or call-chain based predictors [84]. Although using type as a predictor is not as precise as using the call-chain, it has the advantage of imposing much less overhead on the system since determining the type of an object is relatively cheap at run-time. Potentially, this limited overhead can be made virtually free if type-specific allocators are implemented and the compiler inserts calls to the appropriate allocator based on the type of the object to be allocated. If the survival ratio of objects of a given class type is higher than a specified threshold, then future instances of the same type will be allocated in the mature space (pretenured).

Huang et al. combine two dynamic feedback techniques, namely *Jumpstart feedback* and *Continuous feedback*. The first technique, based on the idea that some programs have very consistent survival ratios, gathers life-spans of objects from each class during the first few GC cycles. This information is then used to

drive later object allocations.

They use the second technique to tackle programs that do not have consistent survival. In this scheme, they continuously collect information during each GC and update allocators with such information. Continuous feedback has a higher overhead than the Jumpstart feedback technique. In order to track object lifetimes, they gather per-class statistics at each garbage collection, removing the need for weak references to track object lifetimes. This alternative is cheaper than weak references for two reasons. First, there is no constant sampling overhead since sampling is performed at collection time only. Second, it is more space-efficient because there is no need to record every weak-reference in a data-structure. However, because sampling is performed during GC, longer pauses may be introduced that will have to be offset by the time savings offered by pretenuring.

They show improvements of up to 37% in GC time, and up to 28% in execution time. Unfortunately, they report results using the Jikes RVM BaseBase compiler, rather than the optimising compiler which may amplify the benefits of pretenuring. It is therefore difficult to judge the efficiency of their scheme since researchers usually publish results obtained using the optimising compiler, and it is very likely that if they did so, the performance improvements they observe would be reduced. Furthermore, they only report results from 5 benchmarks (*Jess*, *Javac*, *Jack*, *GCBench* and *GCOld*) which are known to yield a high number of long-lived objects.

2.4.6 Dynamic Object Sampling For Pretenuring

Jump et al. [96] propose a collector with built-in dynamic object sampling where sampled objects are tagged. Their sampling mechanism samples objects every 2^n bytes of allocation and piggybacks on a bump-pointer allocator. At every allocation, the bump-pointer allocator checks whether the allocation exceeds some boundary. If not, the object is allocated: this is the allocation *fast path*. If yes, the allocator executes the allocation *slow path* which normally just checks if more memory needs to be requested, or if a GC should be triggered. At this level an

extra check is inserted to see if the object being allocated should be sampled or not. If the object needs to be sampled, the new allocation *sample path* is taken.

Each sampled object is marked with an additional word that identifies the object's allocation site and an allocation counter is incremented. They note that since a four byte allocation site identifier is added to each sampled object, space requirements are increased by at most 0.8% for a 512 byte sample rate, and 1.6% for 256. During a GC, the collector finds any surviving sampled object and computes a survival ratio for each allocation site.

Their implementation yields some interesting properties. First, by not using weak-references, they avoid having to trace both dead and live objects, hence reducing overheads. Second, by also sampling the actual lifetime of pretenured objects, they can efficiently adapt to program-phases. This secondary sampling mechanism is performed by periodically allocating objects of a pretenured site into the nursery for one allocation phase, allowing the system to reassess the prediction.

The system has a lower time and space requirement than Harris's, because they do not keep track of all sampled objects using weak-references. Rather, they discover sampled objects that survived GC during tracing. This sampling algorithm has the same drawback as Harris's because sampling is performed when a certain boundary of allocation is crossed, which is more likely to happen when allocating large objects.

However, their experiments show poor results as they only improve GC time in one of eight benchmarks, and degrade throughput by an average of 3% when sampling objects every 256 bytes.

In general, sampling-based techniques have the downfall of not handling long-lived objects allocated at the beginning of the program. Unfortunately, evidence shows, in programs like Jikes RVM[4], that a large fraction of long-lived objects are indeed allocated at the start of the program [22].

[4]Jikes RVM is a JVM on top of which a Java program is executed, but since it is written in Java and shares the same heap as the program, is is here considered as a program.

2.4.7 Pretenuring for Java

CHL performs pretenuring decisions based on the fraction of objects allocated at a specific site that survive a minor GC. Blackburn et al. [26, 22] extend this approach. They remove implementation dependency by normalising object lifetime as a multiple of *live size*, which is the maximum volume of objects live at any time in the run of the program.

With an instrumented virtual machine, they record traces of programs in order to analyse them. Using these trace files, they compute the lifetime of each object and classify each object as *short-lived*, *long-lived* or *immortal*. They observe that objects which will never be copied have a lower space requirement than objects that may be copied because they do not need a copy reserve. In an Appel-style collector (see Section 2.1.3), the copy reserve space overhead is half the heap (100% overhead). Therefore, they establish that if the object is going to live longer than the time elapsed between its death and the end of the program, more space would be saved by allocating it in the immortal region, which does not require a copy reserve.

Their classification scheme works as follows:

- An object dying later than halfway between its time of birth and the end of the program is classified immortal.
- Otherwise if the object's age less than a certain threshold[5], it is considered short-lived.
- In all other cases, the object is considered as long-lived.

Based on the fraction of objects of each kind allocated, allocation sites are split into 3 categories: generating predominantly short-lived, long-lived or immortal objects. Given an allocation site that allocates a fraction S_s of *short-lived* objects, L_s of *long-lived* objects and I_s of immortal objects, they classify sites using homogeneity thresholds H_{if} and H_{lf} as follows:

[5]They use $0.45 \times$ *max live size* in their experiments.

- 1. If $I_s > S_s + L_s + H_{if}$, the site is classified *immortal.*
- 2. Else, if $I_s + L_s > S_s + H_{lf}$, the site is classified *long.*
- 3. Otherwise, the site is classified *short.*

The homogeneity thresholds determine the degree of conservativeness: the higher H_{if} is, the more conservative will the predictions be with regards to *immortal* objects. Likewise, the higher H_{lf} is, the more conservative the predictions regarding *long-lived* objects will be.

Wrong predictions can be split into two categories: *Type 1* errors and *Type 2* errors. Type 1 are those that classify a long-lived (or immortal) object as young. The more conservative one is, the more likely these errors are. However, Type 1 errors do not incur any additional cost because they are the default behaviour of generational garbage collectors.

Type 2 errors, on the other hand, are those that classify short-lived objects as long-lived or immortal. These errors are costly because they increase the amount of garbage in older generations, reducing the space available for the nursery, and potentially increase the number of full-heap GCs. Therefore, in the context of this thesis, being conservative means minimising Type 2 errors.

Predictions based on the analysis of a previous recording program are called *program-specific* predictions, or *self-predictions.* For instance, Harris [73] records allocation sites that consistently allocate long-lived objects using an instrumented VM, and later reuse this information to perform pretenuring. This is self-prediction. To a certain extent, this is also the case for CHL [84] and Jump et al. [96] because they first sample the lifetime of objects based on specific criteria, respectively the type of the object and the allocation site during the execution of a program. Once they have some confidence that a certain criteria is associated with a long object lifetime, they can pretenure objects exhibiting this criteria.

Up to this point, Blackburn et al.'s study has investigated self-prediction, where the analysis of past runs of a program is used to predict its future behaviour. However, by analysing a variety of benchmarks, they are able to isolate allocation

sites common to these programs and generate lifetime predictions. By using this advice, programs never encountered before that make use of any of these common allocation sites can benefit from pretenuring (this is true-prediction). Common allocation sites can be found in libraries or even the virtual machine itself if the VM is written in Java. Furthermore, because they use Jikes RVM, which is written in Java, they can make the VM itself benefit from pretenuring, by burning the advice into the compiler (*build-time* advice).

For each trace t, they associate a weight w_t with each site where

$$w_t = v_s/v_t \tag{2}$$

and v_s is the volume allocated at the site and v_t is the total volume of objects allocated in this trace. For each site s, they then generate combined bins $S_{c,s}$, $L_{c,s}$, $I_{c,s}$ using weighted averages for all sites. Using $S_s(t)$ as the value of short-lived objects generated at the site s for trace t, and using

$$w_c = \sum_{t=1}^{n} w_t \tag{3}$$

they calculate the fraction of short-lived objects using the following formula:

$$S_{c,s} = \Big(\sum_{t=1}^{n} S_s(t) * w_t\Big)/w_c \tag{4}$$

Fractions of long-lived objects and immortal objects are calculated in the same manner. Finally, they re-apply their classification algorithm using these new fractions of short-lived, long-lived and immortal objects to classify sites as short-lived, long-lived or immortal.

For each individual program that has been analysed, an advice file matching allocation sites with pretenuring advice is created.

In their experiments, they use an Appel-style generational collector (see Section 2.1.3) with a separate immortal space that is never collected. The mature space is managed by a semi-space collector. After each nursery collection, all

surviving objects are copied into the old-generation.

Their VM implements a command line option allowing loading of a program-specific advice file. During allocation, objects believed to be short-lived are allocated in the nursery. Objects born at sites with long-lived advice are pretenured into the mature space, while objects born at sites classified as immortal are allocated into the immortal space.

Their experiments show impressive speedups in both GC time and execution time, for both build-time advice and program-specific advice. Combining their build-time advice and program-specific advice, they are able to achieve GC-time improvements of between 40% to 70% on average for most heap configurations. Further, total execution time improves on average by 36% for a tight heap.

Their paper is extremely interesting at many levels, and forms an important base on which this thesis builds. First, their paper shows how object lifetimes can be captured independently of GC specific configurations. Second, their classification allows for impressive speedups in garbage collection time. Since they normalise advice with respect to total allocation for a specific execution, they can combine advice from different applications that share allocation sites. This build-time advice helps in reducing GC time further. Note that their speedups are reduced to about 5% when immortal objects are pretenured into the mature space instead of the immortal space.

It was shown that using self-prediction can lead to great speedups. However, the major downside of self-prediction is that it requires recording and analysing large trace files. Gathering trace files and generating advices can take days, rendering the process very impractical to the end-user. On the other hand, by computing a simple offline static analysis of the program, true prediction such as type-based predictions could provide important speedups.

2.4.8 Conclusions

Garbage collection is a fundamental feature of most modern programming languages. Many new algorithms and optimisations have been introduced over the

years, and garbage collection remains an important area of research. It was acknowledged that generational collectors often yield good performance, but the performance of these algorithms can be greatly diminished if the mutator does not follow the weak generational hypothesis which states that "most objects die young". The potential gains that can be achieved by performing pretenuring, an optimisation technique consisting of allocating objects believed to have a long lifetime directly into the mature space in order to save copying time in these collectors were reviewed.

The major downside of pretenuring is the need to record and analyse trace files, a very time-consuming task. This thesis shows how data-mining techniques can help to predict object lifetimes based on code-level static properties, and hence provide true predictions. The following section reviews what data-mining is, how it works, and how it can be used.

2.5 Extracting Knowledge From Data

In recent years, with the dramatic increase in computer storage capacity, gathering data has become standard practice. Advances in modern experimental and observational methods coupled with the amount of scientific data gathered and its complexity have grown massively over the past years. Many sectors such as finance, astrology and biology have acquired very large amounts of data. For example, banks might gather information such as consumer spending, age, sex, marital status, occupation and so on. Biologists on the other hand have gathered large amounts of data such as the human gene pool.

Gathering data is essential to all these sectors because of the information that can be derived from it. By gathering and analysing data, one can predict future events with a certain probability. For instance, banks gather statistics about each type of customer and give or refuse loans based on the probability that this type of customer would repay. By analysing the gene pool, biologists can predict which individuals are susceptible to develop specific diseases based on their DNA.

But extracting knowledge from millions of records is difficult. This section presents the characteristics of the dataset, before reviewing the different methods one could employ to analyse it. The following section focuses on the different techniques and the background necessary to understand the content of this thesis. This thesis does not research on knowledge extraction from data, but rather uses existing research and tools for the purpose of object lifetime prediction.

2.5.1 Problem Characteristics

Extracting knowledge from non-trivial data is a complex problem. Choosing the appropriate technique depends on the dataset and the kind of knowledge one wishes to extract.

Before describing the available techniques that could be used in this context, the dataset is described with respect to object lifetime prediction. The goal is to find correlations between the life span of an object in memory and different

characteristics of a program, in an attempt to predict how long each about-to-be-allocated object is likely to live (categorised as short-lived, long-lived or immortal). For a more complete description of the dataset with regards to lifetime, pretenuring, and more generally garbage collection, refer to Chapter 3.

The goal is to be able to predict the lifetime of an object based on many different attributes (see Chapters 5 and 6).

Available Data

In this thesis, the inputs (or fields) and outputs have different types. In terms of lifetime (the output), the data is *categorical*: following Blackburn et al., objects are divided into 3 different categories (or classes), namely 'short-lived', 'long-lived' and 'immortal'. Inputs, however, can be of different types such as real numbers, boolean numbers, ranges or categories. In Chapter 5, where *software metrics* are used as predictors, 16 inputs of real type corresponding to different measurements and particularities gathered from the source code of Java programs are used. In Chapter 6 however, where *micro-patterns* are used to drive the predictions, 60 input fields of type boolean are used to answer the question "does the class of this record exhibit micro-pattern X".

Problem size

The used training dataset contains many records (29,234) corresponding to the number of allocation sites gathered from the training set (see Chapter 3). Up to 60 attributes are used for each record from which predictions are generated.

Goal

In a pretenuring system (see Section 2.4), allocating a long-lived object in the nursery is not very costly, because this is the default behaviour of a traditional generational algorithm. However, it is shown that allocating a short-lived object into the mature space or into the immortal space can be very costly. For this reason, only those objects that have a very high probability of being long-lived or

immortal are pretenured. Furthermore, this work strives to understand the decision process in order to draw unexpected, yet useful lessons from it. A correlation between some specific object characteristics, or combinations of characteristics to specific lifetimes is sought.

Therefore, a method capable of generating rules such as *if A and B then C*, and associate a probability with each rule is needed. The most accurate prediction rules are then selected to minimise the misclassification rate and apply these to the data. The result of this procedure would be the prediction of object lifetimes, classified as 'short-lived', 'long-lived' or 'immortal'.

2.5.2 Data Mining: A Solution To Many Problems

Traditional analytic methods require a sound mathematical and statistical knowledge and do not scale well to very large datasets [70]. Analysing vast quantities of data gathered by modern techniques calls for a computer-based analytical method.

Data mining has been described as "the nontrivial extraction of implicit, previously unknown, and potentially useful information from data" [58]. It uses real-world data and is a popular choice in industry where large quantities of data are usually abundant. Used appropriately, it can bring valuable insights and aid decision making.

Over the years, many data mining algorithms have been used and developed [152]. They are used to solve many different problems, and each algorithm has its own purpose, strengths and weaknesses. The basic principles of the most common data mining problems are explained below.

The Most Common Data Mining Tasks

Regression consists of finding a function capable of predicting the real value of a variable based on available data. Linear regression, which uses the formula of a straight line ($y = \mathbf{a} \cdot \mathbf{x} + b$), is the simplest form of regression. It is capable of finding the appropriate values for a and b, for which y can be predicted given x.

This is useful for problems such as predicting the yearly spending of a household based on factors like the number of children and the income of the parents.

Clustering tries to group similar data items into clusters based on the values of the attributes that describe each item. Clusters are usually created using a similarity measure which calculates how similar two data-items are to each other. The output of a clustering algorithm is a finite set of categories or clusters that describe the data. An example of a clustering problem could consist of identifying a category of consumers that tend to default on their loan repayments.

Change and Deviation Detection is used to discover significant changes in the data from a previous analysis. For example, meteorologists might use change and deviation detection to visualise major changes in climate from month to month or year to year.

Association Rules are a set of rules derived from the dataset. They are associated with a confidence, and all rules with a confidence greater than or equal to user-specified thresholds are extracted. These rules are of the form "*If attribute X then attribute Y*". For example, supermarkets often group products next to each other based on objects frequently bought together. Association rules can help them discover these relationships.

Dependence Modeling is a data mining task that involves the discovery of dependences among attributes. It helps to relate attributes with each other. Dependence modeling differs from association rules in that it allows the modeling of more complex data. Dependence models are of the form "*If a given set of conditions are satisfied for data X, then predict the value of the goal attribute for data X*" For instance, a bank might find that there is a strong relationship between the salary of the customer and their age.

Classification is the task of finding a model capable of predicting the class of a data item amongst several predefined classes based on values of attributes of that data item. While clustering tries to group output data into relevant categories, based on input data, in a classification task, categories (the output) are already given to the classification algorithm. In other words, clustering allows you to

identify categories of data (categories of customers for example), and classification allows you to make decisions based on categories.

For example, a bank might wish to issue or refuse a loan based on the income of the client their total debt and their age. In this case, the income, debt and age of the customer would be the attributes, and issuing or refusing the loan would be the class.

Data Mining In The Context Of This Thesis

In this research, only 3 classes of objects are considered with respect to their lifetime (short-lived, long-lived or immortal): this is the output. A variation of inputs is used to predict the lifetime of an object from it. Therefore, this is a *classification* problem. The following section, reviews classification algorithms in more detail.

2.5.3 The Classification Task Of Data Mining

For a specific dataset, classification involves finding a model to predict the class of a data item based on its attributes.

The most well known data mining classification models are *Decision Trees and Rules*, *Support Vector Machines* (SVMs), and *Neural networks.*

Algorithms such as SVMs and neural networks have a black-box model, while decisions trees and rules have a white-box model. In a black-box model, the data item's class is predicted without giving the researcher any indication of the way this prediction was achieved. A white-box model instead offers much insight into the decision making process to the researchers. Researchers can visualise the process, and they might discover unexpected relationships which can be of interest.

For this research, it was decided that a white-box approach is more suitable, because this work seeks to understand the decision process and learn from it. For instance, finding relations such as "*If the object is of size X, then it's long-lived*"

would be useful to the GC community.

Therefore, SVMs and neural networks are discarded from the studied set because of their black-box model. *Decision tree* models, on the other hand, allow the understanding of the decision process and can easily be applied to the dataset. *Decision tree* induction algorithms are reviewed in the following section.

2.5.4 Decision Tree Induction Algorithms

Decision tree induction algorithms create classification models represented in the form of a decision tree (which can be converted into a set of rules, as explained below) and can be used to understand the decision making process involved in reaching each classification. A decision tree can have many internal nodes (non-leaf), where each internal node is labelled with the name of an attribute. In decision trees, each arc starting from an internal node represents a possible value of the input variable labelling that node, and each leaf node represents the value of the class to be predicted for any data item that has the variable values specified along the path from the root node to that leaf node.

For instance, to predict the quality of a strawberry (classes 'Sweet' or 'Not sweet') based on its concentration of sugar ('Low' or 'High') and its concentration of water ('Low' or 'High'), decision trees are suitable. Figure 1 shows a hypothetical classification model in the form of a decision tree.

In this model, 'Sugar Level' and 'Water Level' are nodes, 'Low' and 'High' are arcs, and 'Sweet' and 'Not Sweet' are leaves.

Building a decision tree model requires the notion of information gain, also called *Kullback-Leibler* divergence [101]. The information gain value is the reduction in entropy (a measure of the uncertainty associated with a random variable) from a prior state to a new state.

For example, if one was to guess a number between 1 and 1000, he would probably start by asking "is the number less or equal to 500?". Most people would do so intuitively, because this question splits the dataset into two equal parts, hence providing the most information gain. The entropy of the dataset at

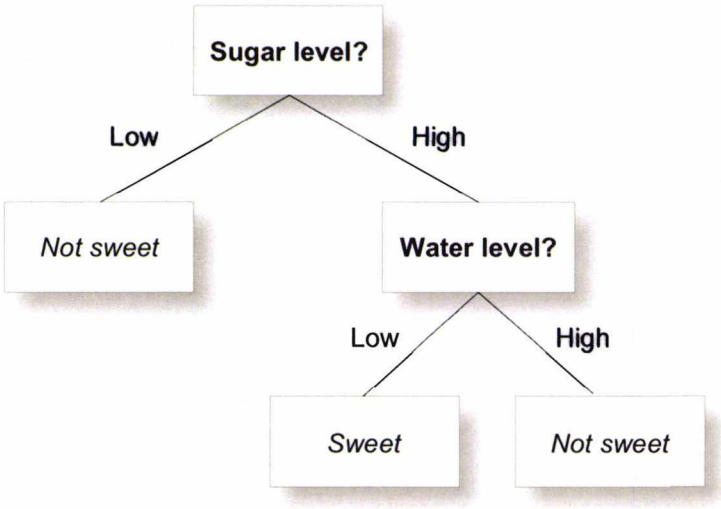


Figure 1: Decision tree for determining the sweetness of a strawberry

the beginning is 10, because we need to ask 10 questions to be sure to find the answer. Once the dataset has been split by the above mentioned question, the entropy of either remaining part is 9. A decision tree model is built with the following steps:

1. Calculate the information gain value for each attribute.
2. Select one attribute with the highest information gain and create a node for that attribute.
3. Make a branch from this node for every value of the selected attribute.
4. Partition the dataset in such a way that each new branch is associated with the fraction of data that exhibits the attribute value.
5. Recursively repeat the process from step 1 for each of the newly created branches until no more partitioning can be performed.

Decision tree induction algorithms have the advantage of creating tree models that are easy for humans to understand (provided the tree size is relatively small):

the classification process is easily understandable. This contrasts with other *black-box* like algorithms where understanding the classification process is difficult, if not impossible. Decision tree models can also be converted into rulesets. I.e. sets of *if-then* rules. This conversion can be done by creating one rule for each path from the root node to a leaf node so that the number of rules will be equal to the number of leaf nodes.

For instance, the previous example of a decision tree in figure 1 would be converted into the set of rules:

- IF sugar = 'Low' THEN class = 'Not sweet'.
- IF sugar = 'High' AND water = 'High' THEN class = 'Not sweet'.
- IF sugar = 'High' AND water = 'Low' THEN class = 'Sweet'.

Decision tree induction algorithms can also handle different types of data, numerical or categorical, and cope well with large datasets.

For this research, Clementine [64], a well known commercial data mining system which provides many different data mining algorithms in an integrated environment, in a plug and play manner was used.

A summary of the requirements explained above is as follows:

- The model must be easy to understand.
- Rules are needed in order to apply the model easily to the data.
- An algorithm capable of coping with different types of data input, such as ranges, booleans and categories is needed.
- There is no need to be able to predict every object, but predictions with a very high confidence are required.
- Ideally, the algorithm should associate a confidence level with each rule so that only those rules with a very high confidence are selected, in order to limit misclassifications.

Based on these requirements, it was decided to use the *C5.0* algorithm which is the only algorithm available in Clementine capable of meeting all the requirements. Its capabilities are briefly reviewed below.

C5.0

C5.0 [128] was developed by Ross Quinlan as an improvement over the well-known C4.5 algorithm [129]. The latter is a powerful and widely used data mining algorithm designed to analyse datasets containing thousands to hundreds of thousands of records and tens to hundreds of fields. It allows different types of input such as booleans, categories, ranges and real numbers. Classifiers can be expressed as *decision trees* or *sets of rules*, which are easy to understand. Furthermore, this algorithm allows for decision trees to be turned into a set of *if-then* rules, to which confidence levels can be added. C4.5, and C5.0 are popular decision tree induction algorithms renowned for (amongst other things) the quality of their outputs, their flexibility in handling different types of attributes, and their speed.

In a complex dataset, even the best algorithm can not perfectly model all the data items. *Adaptive boosting* [59], a technique implemented in C5.0, aims at further refining previously generated models. It is capable of generating and combining multiple classifiers to improve predictive accuracy. Boosting reassesses the previously generated model, and focuses on data items that have a high misclassification error rate. This process can be reapplied iteratively (in several rounds) in order to further improve the final model. During each boosting round, weights are applied to the data items based on their classification error rate: the harder an example is to classify, the higher its weight. The algorithm will then focus mainly on the most weighted example in order to improve the overall accuracy of the model. At the end of each round, weights are re-evaluated to take account of the latest changes.

2.5.5 Summary

Data mining is a technique capable of extracting potentially valuable information from vast amounts of data. By choosing a white-box algorithm such as a decision tree induction algorithm, information can be easily understood by humans, and lessons can be drawn from it. Using Clementine and C5.0, a large object lifetime database can easily be data- mined, and predictions can be derived. The possibility offered by C5.0 for generating rules and associating confidence levels is particularly useful since only those rules that have a very high confidence level can be used, in order to avoid misclassifications.

The next chapter reviews the methodology employed to gather and data-mine lifetime information, derive predictions from it, and use these to drive allocation inside a specifically modified JVM.

Chapter 3

Methodology

This chapter describes the methodology employed to carry out the experimentation. This methodology was chosen with the goal of maximising the accuracy of the predictions and maximising the potential performance gains. At the same time, this approach is general enough to allow comparison with other research carried out in this area.

Section 3.1 gives an overview of the process from recording trace files to generating a knowledge bank. Section 3.2 describes the experimental platform, while Section 3.3 describes Jikes RVM in detail. Section 3.4 describes the process by which allocation sites are associated with lifetimes before reviewing how this thesis takes advantage of this knowledge to predict object lifetimes ahead of time. Section 3.5, explains how the predictions are made. Section 3.6 discusses how objects are classified with respect to their lifetime. Section 3.7 introduces the lifetime predictors. Finally, Section 3.8 describes the methodology employed to run the data-mining process.

3.1 Process Overview

This thesis discusses how the lifetime of objects within programs never encountered before can be predicted ahead of time. In order to do this, it is necessary to associate certain characteristics of the program with object lifetimes. This section

gives a high-level overview of the process to help the reader visualise the stages involved. Each stage is described in greater depth in the remainder of this chapter.

The main stages involved in the methodology are as follows:

1. Gather object lifetimes statistics by recording and analysing tracefiles from a training set comprising of as many programs as possible (see Sections 3.4 and 3.5).

2. Associate a lifetime to each allocation site, based on the lifetimes of the objects it allocates (see Section 3.6).

3. Discover specific characteristics about the classes of each program within the training set. These class-level characteristics are referred to as *predictors*, because they are used to predict object lifetimes at a later stage (see Section 3.7).

4. Map each allocation site with two sets of predictors: those associated with the class that contains the allocation site, and those associated with the class allocated (see Section 3.7).

5. From the two previous stages, derive mappings between predictors and lifetimes (see Section 3.7.2). In more general terms, this mapping associates characteristics of a source class and a destination class with a lifetime.

6. Data-mine the relation from step 5 and generate rules predicting the lifetime of an object based on predictors. This information is stored into a knowledge bank (see Section 3.8).

7. Determine predictors of a program never encountered before, and query the knowledge bank to predict the lifetime of each allocation site (see Section 3.9).

The remainder of this section highlights the reasoning behind this methodology, the necessary steps towards the creation of a knowledge bank, and this knowledge bank is exploited.

3.1.1 Rationale

The process described allows the prediction of the lifetime of objects within a program never encountered before. This work explores how static program properties, based on source-code, can be exploited to predict the lifetime of objects.

Intuitively, the goal is to understand if specific programming patterns or coding practices have an impact on object lifetimes, and if these can be captured by predictors. Since this work is trying to detect hints from the code itself to predict lifetimes, it is hypothesised that there might be some classes or types of classes allocating exclusively or predominantly objects of a specific lifetime.

Existing state of the art techniques require either gathering and analysing program traces for every program, or are performed on the fly using online sampling. The first technique requires a large amount of time and needs to be performed for each individual program, and each input. The second technique is not able to handle long-lived objects allocated at the beginning of the program, and has a relatively high runtime overhead due to sampling. A more complete discussion about the trade-offs associated with each technique was presented in Section 2.4.

Although pretenuring techniques based on the analysis of trace files are expensive, Blackburn et al.'s scheme offers important speedups in both GC time and throughput (see Section 2.4.7).

Therefore, if simple program characteristics can give us an indication about program lifetimes, state of the art pretenuring systems can be improved by predicting the lifetime of objects in programs never encountered before at a low pre runtime cost.

3.1.2 Creating A Knowledge Bank

The creation of the knowledge bank is performed in several steps. First, the execution of several programs is recorded and stored into trace files (see Section 3.4.2). Trace files are then analysed in order to associate a lifetime with each allocation site (see Section 3.6). The source predictors and destination predictors

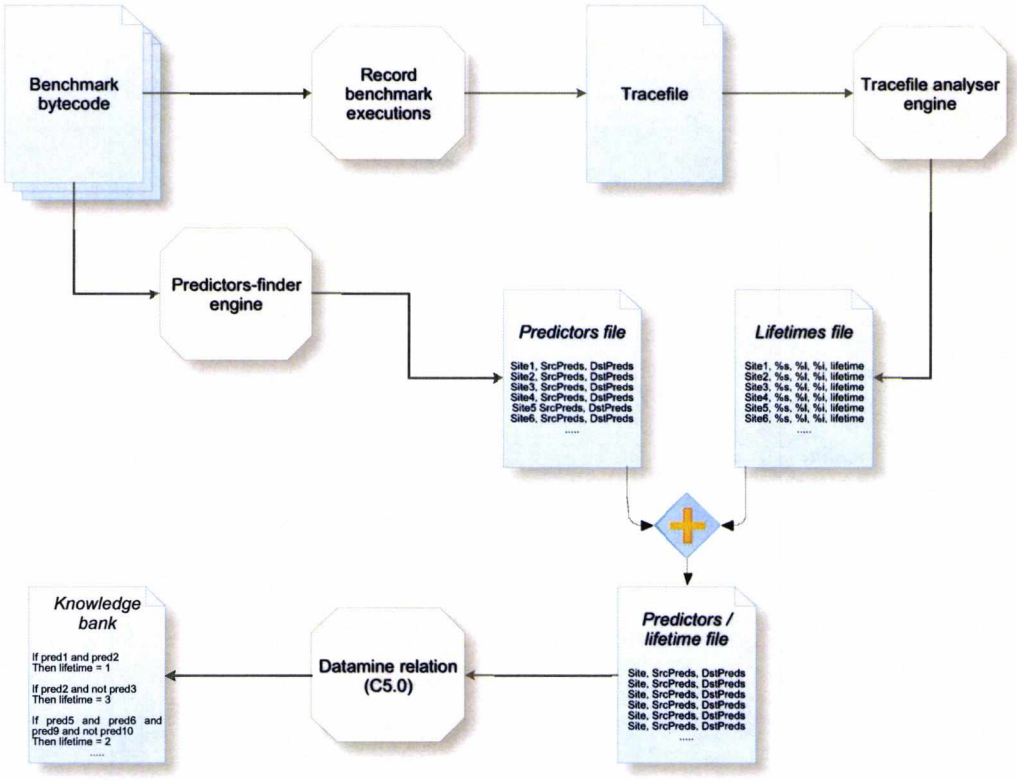


Figure 2: Creating a knowledge bank.

associated with each allocation site is then retrieved, and the lifetime classification gathered previously (see Section 3.7.1) is then added. Finally, this information is data mined and rules matching predictors with lifetimes (see Section 3.8) are generated. Figure 2 summarises this process.

3.1.3 Exploiting The Knowledge Bank

Exploiting the knowledge bank is a simple process. First, predictors are extracted from the program to be run. Then, these predictors are matched against the knowledge bank and a lifetime advice file is generated. Finally, the JVM loads the advice file and executes the program, pretenuring when necessary. Figure 3 summarises this process.

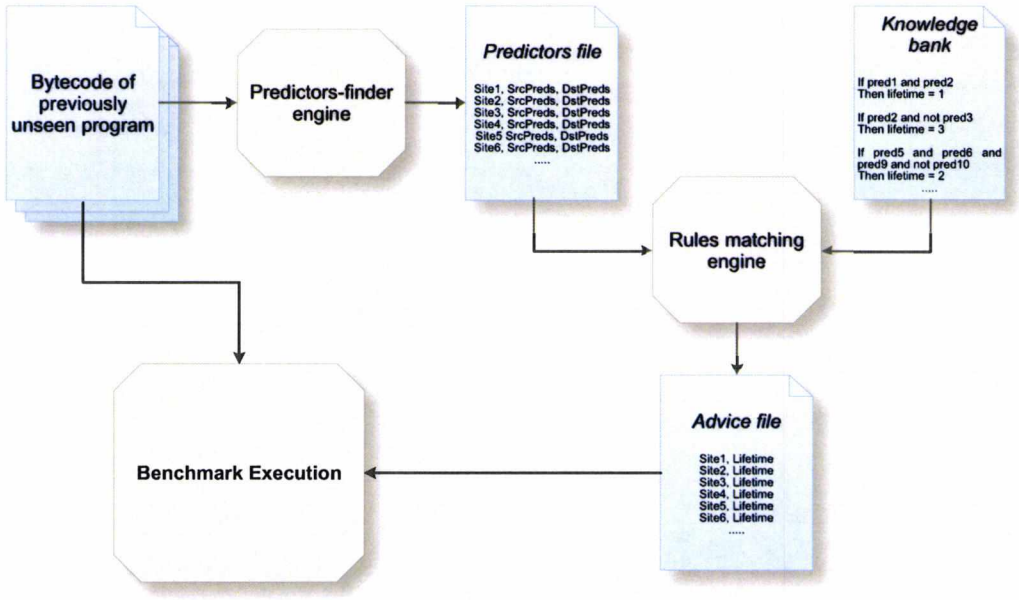


Figure 3: Using the knowledge bank.

3.2 Experimental Setup

The following list describes the hardware platform in which both training and testing experiments were performed:

- Model: Dell Optiplex GX270.
- Processor: Intel Pentium 4 2.6 GHz processor with 800 MHz front side bus and Hyper-Threading.
- Level1 Cache: 8 KB.
- Level2 Cache: 512 KB.
- Memory: 1 GB (dual-channel shared DDR SDRAM, 333MHz).
- Operating System: Debian GNU Linux, 2.6.12 kernel.
- Jikes RVM version 2.4.4 (described in Section 3.3).

3.3 Jikes RVM

Jikes Research Virtual Machine (Jikes RVM) [1] is an open source virtual machine developed by IBM, previously known as the Jalapeño virtual machine. In 2001, Jikes RVM was donated to the open source community in an attempt to facilitate research and offer a standard experimentation platform where novel virtual machine ideas could be explored, measured and evaluated. Today, Jikes RVM is the most popular experimentation platform in the GC research community and is well supported. Programmers can modify and experiment with every part of the system.

3.3.1 A Java Virtual Machine Written in Java

One of the main attractions of Jikes RVM is its interesting property of being a Java virtual machine written in Java. Commercial JVM implementations often favour a lower-level language such a C or C++. However, Jikes RVM was developed for research purposes and as a proof of concept to show that a Java virtual machine could actually be written in Java. Because the implementation is Java based, it offers some benefits. For instance, it allows the use of an automatic memory manager within the JVM itself, therefore reducing the potential for memory leaks and preventing dangling pointers. It also offers type safety as well as object oriented design and other benefits offered by the Java language.

Although most of Jikes RVM's code is implemented in Java, the implementation must sometimes evade the restrictions of the Java language through Jikes RVM *Magic*. *Magic* methods are used by Jikes RVM's various compilers to perform operations such as raw memory access, perform unsafe casts or trigger operating system calls which cannot be implemented in Java code. Despite this necessity, users remain restricted to the limitations of the Java language in order to ensure code safety. For critical sections of the JVM, such as interfacing with the operating system, Jikes RVM uses some C code (*glue* code). The proportion of C code in the overall implementation is small (less than a few thousand lines).

3.3.2 Boot Loader

Because Jikes RVM is written in Java, it requires a substantial set of services — a class loader, an object allocator and a compiler — before even being able to run its own code and start the complete virtual machine. The *boot-loader* is in charge of loading the *boot-image* which encapsulates all the essential core services for Jikes RVM to run [146]. Later, the boot-image is loaded into memory and executed before running the Jikes RVM itself.

The boot-image itself is created by an external *boot-image writer* based on the existing Jikes RVM code. Part of the C code used in Jikes RVM has the purpose of implementing the boot loader.

3.3.3 Optimising Compiler Configuration

Unlike standard JVMs, Jikes RVM does not use a bytecode interpreter. Instead, Jikes RVM compiles bytecode into machine-code using a Just In Time (JIT) compiler. Researchers can choose (and experiment) between several JIT compilers, including an optimising compiler, which comprises several levels of optimisation.

The optimising compiler is in charge of increasing the efficiency of *hot* methods (methods that are frequently accessed) by recompiling them at a higher level of optimisation. Recompiling a method at a higher level of optimisation is expensive, and the adaptive system ensures this operation is done asynchronously [33].

To achieve performance speedups, the optimising compiler performs aggressive method inlining (replaces a function call site with the body of the callee), increasing the size of the program instruction set. To optimise hot methods, the optimising compiler itself needs to allocate temporary data, resulting in an increase in the program's memory footprint. The induced side effect for this work is that the lifetime of objects allocated by the mutator is artificially increased (remember that time is measures in number of bytes allocated).

3.3.4 MMTk

Jikes RVM uses a very well defined and highly reusable memory management toolkit called MMTk [21]. This module is now part of the standard distribution of Jikes RVM.

This framework is flexible because it allows the quick implementation of new garbage collectors. Its reusable components offer a common base for the main garbage collector algorithms. Its modularity and portability allow it to be easily reused in other virtual machines.

MMTk natively supports many garbage collectors such as *copying collectors*, *mark-sweep*, *reference counting* and different flavours of *generational* collectors. Researchers wishing to implement a new garbage collector can usually extend an existing collector.

MMTk also implements several *allocators* to suit different purposes: a *bump-pointer* allocator, a segregated *free-list* allocator and a *treadmill* allocator [11]. The *bump-pointer* allocator allocates objects in a sequential manner where the next memory slot available is found by incrementing the *bump-pointer* by the size of the last allocated object. The *free-list* allocator divides memory into blocks of different sizes. In a *treadmill* allocator, objects are allocated by moving the *free* pointer forward in a doubly linked list [11]. A size-segregated *free-list* maintains references to the available memory blocks. New objects are then allocated in a block of memory which accommodates the size of that object.

As a results of its well defined structure and enthusiastic community combined with its efficiency and ease of use, Jikes RVM is the ideal Java platform to perform experiments and advance the state of the art in areas such as dynamic compilation, adaptive optimization, garbage collection, thread scheduling and synchronization.

3.4 Discovering Object Lifetimes

Knowing ahead of time when an object is likely to die is a very appealing idea (see Section 2.2). Appropriate exploitation of this information can significantly reduce

the time spent tracing objects, hence reducing GC pause times [22]. This section discusses the compiler setup used to gather object lifetimes, before discussing how trace files are gathered and analysed to discover object lifetimes.

3.4.1 Optimising Compiler Setup

Section 3.3.3 described how the optimising compiler can help improve performance. However, in order to optimise hot methods, the optimising compiler needs to allocate temporary data.

Because the focus is on application allocated data, the wish is to minimise the effect of compiler-allocated data exaggerating object lifetimes. Therefore, the training data set is profiled with the "BaseBase" configuration. In this configuration, the compiler does not perform any type of optimisation either at build-time, or run-time. Since this thesis is not concerned about performance at this stage, using the BaseBase configuration is the best choice for measurement purposes as objects lifetimes can be captured with more precision.

3.4.2 Recording Trace Files

In order to analyse object lifetimes, object information such as the time of allocation and death is recorded into files referred to as *tracefiles*. This process is explained below.

Time

As explained in Chapter 2, measuring the time during which an object lives can prove difficult. This is due to the non-determinism nature of the JVM itself and on the operating system underneath.

In a JVM, many things can happen at different times from one run of a particular input and settings to another. A list (by no mean exhaustive) of the main causes of non-derminism in the JVM is listed below:

- VM thread switches typically happen at different times, and depending on the place in the code at which they occur, can extend or decrease the program's runtime. A typical example happens when threads have to synchronise.

- JIT compilation happens at different times depending on how "hot" a method is (see Section 3.3.3), which in turns depends on how many times the method has been executed.

- Garbage collections also happen at different times depending on the amount of free memory available in the heap. This in turns depends on several factors including the amount of data allocated by the JIT compiler which can be triggered at different times.

- Cache also plays an important role because depending on the code being executed, the cache can get filled with relevant or irrelevant data.

On top of these, the operating system itself can cause non-determinism with threads switching at different times, allocating more or less time for the JVM to run.

Because of these issues, the time spent by the program, the GC, or the lifetime of an object is highly variable if measured in wall clock time. Instead, it is common practice in the world of garbage collection to measure time in terms of bytes allocated.

Unfortunately, this measure is not perfect because it is still non-determinate in the presence of threads. Also, an optimising JIT compiler allocates data to perform compilation, therefore artificially increasing the lifetimes of user program objects even if all user threads are stopped. Note that as described in section 3.10.9, the BaseBase compiler configuration is used for the measurements, which does not perform any type of optimisation either at build-time, or run-time.

Recording Allocations

In Jikes RVM, each object is allocated via a method known as the allocator. This method is in charge of allocating objects in the relevant part of the heap depending on allocation policies. For instance, immortal objects are allocated in the immortal region while large objects are allocated in the *Large Object Space*. In the case of a generational collector, all other objects are allocated in the nursery.

In order to create trace files, the allocator is modified so that each allocation is recorded into a trace file. However, a more subtle mechanism is required to record the time of death of an object.

Merlin

Recording the exact time of death of an object is ideal. However, a brute force approach consisting of triggering a full-heap GC at every allocation could take a simple program months to run.

Merlin is a trace generation algorithm that was developed to allow researchers to gather exact lifetime data in a shorter period of time [75]. Unlike a brute force approach, it does not trigger extra GCs (apart from those GCs generated by the fact that Merlin itself generates some garbage). Instead, Merlin timestamps live objects whenever they may become unreachable. After each GC, Merlin finds out which objects are dead, and reconstructs the time of death of each object, based on the last timestamp. If a dead object has a timestamp older than one of its descendents, then the timestamp is propagated to its decendents, since they had to be live at the time that particular object was last timestamped.

Unfortunately, gathering traces with Merlin can still be slow. For example, profiling *javac* from the jvm98 benchmarks suite with Merlin can take a week whereas a non-instrumented run of the program would usually be completed within less than 10 seconds.

MemTrace

MemTrace, developed Richard Jones and Chris Ryder [94], performs a full garbage collection at regular intervals (e.g. 64 KB) and records dead objects in a trace file after each garbage collection. Garbage collection is done using a semi-space garbage collector as MemTrace requires all GCs to be full heap collections. Internally, MemTrace keeps a bitmap matching each entry to a specific memory location. Whenever a new object is allocated, a bit corresponding to its location is set in the bitmap.

During a GC, surviving objects are copied to *to-space*. The bits associated with their old location are then unset, and new bits are set to record their new locations. At the end of the GC phase, the *from-space* part of the bitmap is scanned, and every object with an associated bit still set is considered dead, and a new death record is created.

The Jikes RVM compiler is modified to cause the object ID (which corresponds to its time of birth) and the allocation site ID (which is mapped to the object's allocation site in a separate file) to be written into the header of each allocated object. MemTrace can therefore output death records containing the object ID, the age of the object and the allocation site ID that created the object.

Unlike Merlin, MemTrace is capable of profiling *javac* in 3.5 hours. On the other hand, because MemTrace performs garbage collections every 64 KB of allocation, object death times are only accurate to 64 KB.

Although a granularity of 64KB tends to exaggerate the lifetime of short-lived objects, it c an be argued that no practical garbage collector can take advantage of a finer granularity. The trace files analysed in this thesis were recorded using MemTrace and acquired from MemTrace's creators.

Size of a Trace File

MemTrace's recorded trace files usually contain hundreds of megabytes of data, and can sometimes consume several gigabytes uncompressed. In comparison, Merlin's trace files are much bigger as they also contain records about each pointer update.

3.4.3 Inside a Trace File

Structure of a Trace File

The MemTrace trace files have the following line-based format:

- Allocation record: *A objectID size siteID*
- Immortal allocation record: *I objectID size siteID*
- Death record: *D objectID size age siteID*

The *objectID* is an identifier unique to each object, and the *siteID* is an identifier unique to an allocation site. Size and age are recorded in bytes.

Note that while the death record contains the age of the object, recording allocations is still necessary because some objects do not die.

Once trace files have been recorded, they are analysed, and the lifetime of each object is computed. Likewise, each allocation site needs to be associated a lifetime based on the lifetimes of objects allocated at each site. Then, for each program and each input, a unique file is created which associates allocation sites with lifetime.

The next section discusses the lifetime classification policy.

3.5 Training Data Set

The previous section discussed how trace files are gathered and analysed. This section reviews the training set for data mining, from which trace files are gathered.

Benchmark	*Files*	*NC LOC*
DaCapo Package	268	42244
DaCapo Shared	24	3789
DaCapo Shared Ratio	8.95%	8.97%
jvm98 Package	177	58873
jvm98 Shared	46	6006
jvm98 Shared Ratio	26%	10.2%

Table 1: Harness statistics for DaCapo and jvm98

The purpose is to explore the extent to which program-specific prediction can be provided to guide pretenuring. However, it is important that such program-specific prediction nevertheless be true prediction rather than self-prediction [14], that is, the prediction should not be derived simply from a past execution of the same program (see Section 2.4.2).

The next section discusses the *Cross Validation Leave Out One* technique in the context of this work's training set.

3.5.1 The Case for CVLOO

Cross Validation Leave Out One [100] is a common machine learning technique which consists of dynamically splitting the dataset into a training set and a test set until all combinations have been covered. For example, given 10 benchmarks, CVLOO would train on 9 benchmarks and test on the remaining one. Each possible combination of training set and test set is tested before generating results. It was decided not to use CVLOO to be sure that no element of self-prediction would be found in the methodology.

For instance, all benchmarks of jvm98 share common code such as the test harness, input/output (IO) code, reporter code, as well as libraries. In the same manner, the *DaCapo* benchmarks (see Section 3.5.2) also share common code.

Table 1 shows the quantity of code that is shared between benchmarks, excluding libraries. In this table, "DaCapo Package" and "jvm98 Package" refer to the source code of each set of benchmarks, excluding libraries. "DaCapo Shared"

and jvm98 Shared show how much of the code within each package is shared. "DaCapo Shared" Ratio and jvm98 Shared Ratio are the ratios of shared code when compared to the whole package. The second column "Files" is a count of the number of corresponding files and "NC LOC" is the non-comment lines of code. As can be seen from the table, excluding libraries, the *DaCapo* benchmarks share 8.97% of lines of code, and the jvm98 benchmarks share 10.2%.

What is more, the *DaCapo* benchmarks are all real-world open-source applications that share a large amount of library code [20]. If the CVLOO approach had been chosen, there would have been the risk of allowing some code to be available both in the training set and the test set.

Instead of using CVLOO, benchmarks are divided into a training set and a testing set. Training is performed using the *DaCapo* benchmarks, a set of benchmarks gathered from real-world applications. Testing, on the other hand, is performed on the jvm98 suite [147] which provides a set of well-tested benchmarks commonly used in the GC field to measure performance. While jvm98 benchmarks are not as large as *DaCapo*, they offer a well tested environment to run programs and allow the comparison of findings with previous research. Using this approach ensures that the training set and the test set are two completely different sets of benchmarks.

3.5.2 DaCapo Benchmarks

As explained above, the analysed benchmarks are the *DaCapo* benchmarks (version 051009), a set of open-source real-world application [20]. This set of benchmarks is referred to as the *training set*. DaCapo was used to construct rule-sets because it is the best representative of 'real-world' programs: it comprises a set of benchmarks with a large number of classes, written in an object-oriented style, generates intensive memory loads, and provides a large knowledge bank for the rule sets. Table 2 reports the maximum livesize of each benchmark, the amount of data it allocates and the number of allocation sites it exhibits for each input size available.

Program	*Input*	*Max. live (MB)*	*Allocated (MB)*	*Sites*
antlr	default	7.09	270.21	2014
antlr	large	8.21	649.6	2014
antlr	small	6.15	19.37	1180
bloat	default	9.90	389.98	996
bloat	large	12.72	3218	1940
bloat	small	10.35	66.62	1696
fop	default	16.47	130.63	2334
fop	large	17.61	132	2336
fop	small	11.81	82.65	2269
hsqldb	default	19.8	1025	1265
hsqldb	small	8.34	227.63	1268
jython	default	7.56	387.85	1464
jython	large	9.02	1724	1432
jython	small	8.86	49.45	1410
pmd	default	15.38	281.91	1384
pmd	large	16.56	1533	1393
pmd	small	9.42	54.84	1358
ps	default	4.12	548.81	1179
ps	large	15.39	1676	1166
ps	small	5.03	157.03	1176

Table 2: The DaCapo benchmark suite, v. 051009, BaseBase compiled. *Max. live* is the largest volume of data live at any point in the program, *Allocation* is total allocation and *Sites* is the number of sites used at run-time by the benchmark, Jikes RVM or libraries.

3.6 Lifetime Classification

In order to predict accurately the lifetime of an object, self-prediction techniques such as those described by Blackburn et al. and Chen et al. (see Section 2.4) require analysing previously recorded data. This step is a crucial in their methodology, and also in this thesis. The approach outlined in this thesis differs from theirs because of the fact that this thesis only need to do this once, whereas they need to trace and analyse any previously unseen programs.

Classifying the lifetime of an object in order to generate accurate pretenuring advice is a difficult task (see Chapter 2.4). Ideally, the classification scheme needs to be generic enough to be applicable to any type of generational garbage collector.

Accurate and exploitable lifetime predictions would allow us to make correct pretenuring decisions, hence reducing the time spent copying data and reducing the overall GC time. On the contrary, wrong lifetime predictions, would fill the mature and immortal spaces with short-lived objects, hence requiring more frequent GCs, or would allocate objects with a long life span in the nursery. It is therefore crucial for the classification scheme to deliver accurate and exploitable pretenuring advice.

3.6.1 Lifetime Classification Scheme

This thesis analyses ways by which the lifetime of an object can be predicted ahead of time. After reviewing the relevant literature (see Section 2.4), it was decided to use Blackburn et al.'s [26, 22] classification scheme (see Section 2.4.7). The reasons for this choice are reviewed below.

Performance

The classification scheme of Blackburn et al. offers impressive performance gains over GC time and throughput. Their scheme has been proven to offer good results across many different conditions (different heap sizes, different programs, different inputs, etc.).

Reusability

They capture and classify object lifetimes independently of GC specific configurations. Therefore, by using this scheme, the predictions could be applied to any generational GC.

Lifetime Groups

Despite important speedups, their scheme uses only three lifetime groups: short-lived, long-lived and immortal. This factor is important because the data-mining algorithm needs only to consider three potential lifetime predictions, hence reducing the possibilities for error (see Section 2.4.7).

Point of Reference

A natural consequence of the choice is that by using the same classification technique as Blackburn et al., their results can be used as a point of reference with which to compare the work carried out in this thesis.

After recording and analysing traces, their scheme classifies *directly* each allocation site into one of three categories. Instead, predictions for each allocation site are generated by matching the predictors at each allocation site against the knowledge bank. Therefore, the new scheme is less precise than Blackburn's since unlike self-prediction, this thesis applies the predictions to previously unseen programs, and this thesis considers self-prediction (direct lifetime prediction from traces analysis) as the best-case scenario. Achieving performance gains close to, or better than self-prediction would be siginificant achievement.

Homogeneity Thresholds

Like Blackburn et al., this thesis classifies objects in three categories *short-lived*, *long-lived* and *immortal*, by using thresholds expressed as fractions of the maximum live size[1] of the program. In the experiments, a homogeneity $H_{lf} = 0.45$

[1]Maximum volume of objects live at any time during the program's execution.

and $H_{if} = 0.0$ are used as these are the thresholds used in their experiments[2]. For a more complete description of the classification process, please refer to Section 2.4.7.

An important consideration for the research of this thesis is the definition of immortal objects. While Blackburn et al. define an immortal object as an object dying later than halfway between its time of birth and the end of the program, this thesis investigates the isolation of predictors that have a direct correlation with a more traditional definition of an immortal object: an object that dies only at the very end of the program. This immortal classification is discussed in the following paragraph.

3.6.2 Immortal Classification

While it is commonly acknowledged [113, 1] that segregating immortal objects in a different space can lead to performance improvements, the definition of an immortal object can be ambiguous. Blackburn et al. [26, 22] define an immortal object as an object dying later than halfway between its time of birth and the end of the program (see Section 2.4.7). This definition of immortal objects is effective in the context of an Appel-style collector since the copy reserve space overhead is 100% (half the heap). In this context if an object is going to live longer than the time elapsed between its death and the end of the program, more space would be saved by allocating it in the immortal region, which does not require a copy reserve (see Section 2.4.7).

While Blackburn et al.'s scheme has proved to work very well, this thesis also investigated a different classification scheme of immortal objects. A more natural definition of immortal, which would apply equally to any collector possessing an immortal space, is for objects that die at the very end of the program. Using this definition, it is expected that less objects would be considered immortal. However, as explained in Section 3.1.1, the intuition was that there might be

[2]Recall that H_{lf} is the homogeneity threshold for long-lived objects, and H_{if} is the homogeneity threshold for immortal objects (see Section 2.4.7).

some programming constructs that can be captured, which correlate with the allocation of objects dying only at the very end of the program.

In general, this thesis found that where pretenuring advice provides performance gains, the benefit is sometimes greater with the heuristic definition (Blackburn) than with the 'true' definition (object dying at the end of the program). Otherwise, results are similar. For this reason, only results using the heuristic definition (Blackburn et al.'s definition) of immortal objects are reported[3].

3.7 Lifetime Predictors

This thesis explores two different types of information about object classes in order to predict their lifetime: software metrics (see Chapter 5) and micro-patterns (see Chapter 6). Software metrics help describing the quality and complexity of a software system in an impartial and objective way. Instead, micro-patterns, are similar to design patterns [60] but closer to the implementation: a set of micro-patterns may implement a design pattern.

Because the following sections of this chapter apply to both approaches, these different types of information are simply referred to as *predictors*. This section assumes that for any class, there is a static analysis that provides a predictor set. In both cases, each class is associated with its corresponding set of predictors.

3.7.1 Source and Destination

In order to gain more information about each allocation site, this thesis not only considers the class of the object it allocates, but it also considers the class of the object that allocated it. In the remainder of this chapter, the following definitions are used:

- *source* refers to the class within which the object is allocated.
- *destination* refers to the class of the object allocated.

[3]Remember that time is calculated in number of bytes allocated (see Section 2.1)

For example:

```
1        public class Foo {
2            ...
3            Bar bar = new Bar();
4            Mumble mumble = new Bar();
5            ...
6        }
```

In the code sample above, a new object *bar* of type *Bar* is allocated within a class of type *Foo*. An object *mumble* of static type *Mumble* and dynamic type *Bar* is also allocated in the same class *Foo*.

Because both allocation sites are located in class *Foo*, the class of type *Foo* is considered as the source, and the class of type *Bar* as destination in both cases. Note that for simplicity, only the dynamic type of an object is considered, and therefore *Mumble* is not considered in the second allocation. Future work could explore the use of static type instead of dynamic type and generics as predictors.

To the best of our knowledge, the only publication considering the use of source object as well as the destination object is the paper *Decrypting The Java Gene Pool* published at the International Symposium in Memory Management in 2007 by Marion et al. [95].

3.7.2 Mapping Sets Of Predictors To Lifetime

The gathering of trace files in order to calculate object lifetimes, and how they are used to associate a lifetime with each allocation site was previously discussed. The association of source and destination classes with each allocation site in order to gather more context was also discussed. This section discusses how all this information is mapped together, in order for the data-mining tool to process it and generate predictions.

Consider a finite set of predictors $\mathcal{MP}$ (table 3) and a lifetime $\mathcal{LT}$. Each allocation site is associated a lifetime and a set of source predictors and destination predictors (mapping $\mathcal{S}$).

$$
\begin{array}{rcll}
\mathcal{MP} & = & \{p_0, p_1, \ldots, p_n\} & \text{(Set of all predictors)} \\
\mathcal{LT} & = & \{short, long, immortal\} & \text{(lifetime)} \\
\mathcal{SL} & = & SiteID \rightarrow \mathcal{LT} & \\
\mathcal{S} & = & SiteID \rightarrow \mathbb{P}(\mathcal{MP}) \times \mathbb{P}(\mathcal{MP}) & \\
\mathcal{PL} & = & \mathbb{P}(\mathcal{MP}) \times \mathbb{P}(\mathcal{MP}) \rightarrow \mathcal{LT} &
\end{array}
$$

Table 3: Site, predictors and lifetime mappings.

Using this information, the mapping $\mathcal{PL} = \mathcal{SL} \circ \mathcal{S}^{-1}$ is needed to associate sets of source predictors and destination predictors with a lifetime. In other words, the association of characteristics of a source class and a destination class with a lifetime is needed.

The aim is to find predictors of the source class, the destination class, or a combination of both, which correlate highly with specific lifetimes. The following section reviews how the data-mining tool can use this information to find predictors that highly correlate with object lifetimes.

3.8 Mining The Data

Previous sections showed how to map sets of source predictors and destination predictors to lifetimes. In order to be able to make predictions regarding the lifetime of Java objects, the site–lifetime relation $\mathcal{PL}$ is data-mined to discover which attributes (combinations of patterns) are good lifetimes predictors.

This section describes the data-mining setup used in this thesis, how rules are generated, and how they are exploited them.

3.8.1 The Dataset

The dataset comprises of 29,234 records which correspond to the number of allocation sites gathered from the DaCapo benchmarks. Each record is composed of a set of source predictors, a set of destination predictors and the allocation site's computed lifetime. Records for software metric predictors contain 16 different inputs (8 source predictors and 8 destination predictors) only, while records

for micro-pattern predictors contain 60 different inputs (30 source micro-patterns, and 30 destination micro-patterns).

Before performing data-mining, classical approaches usually require the researcher to *clean* the data set before feeding it to the algorithm. Cleaning the data set is done to ensure that there are no duplicate entries that may skew the results. This process is usually necessary due to the way data is gathered in domains such as biology.

Unlike the classical data mining approach, it was decided not to clean the data set by removing duplicate or contradictory data. There are two reasons for this choice. First, $\mathcal{PL}$ is one-to-many. Second, this research intends to take account of reinforcement of a prediction (i.e. if many inputs make the same prediction, this strengthens the merit of the prediction).

3.8.2 Boosting

In order to improve the accuracy of the predictions, the boosting feature offered in C5.0 (see Section 2.5.4) was used. This allowed the authors to refine the final model, and optimise the rules. In the experiments, 10 rounds of boosting were performed, allowing the boosting algorithm to refine the classification model to a high level.

3.8.3 Generating Rules

Using Clementine's C5.0 algorithm (see Section 2.5), and using the training dataset results as explained above, the following type of rules were generated:

```
1        //Rule 57 for  IMMORTAL  (351.712,  0.996)
2        if Src=#x and Dst!=#y and Dst=#x then IMM
```

Outputs from C5.0 rules are in disjunctive normal form. The rule above is true in *99.6%* of the cases: this is the confidence level. "Src #x" and "Dst #y" refer to specific predictors used as source and destination predictors respectively. The rule states *"If the source exhibits predictor #x and the destination does not*

exhibit predictor #y but also exhibits predictor #x, then instances allocated by this site are immortal in 99.6% of the cases".

C5.0 thus provides a set of rules associating predictors with lifetime advice and a confidence level. This data constitutes a historical *knowledge bank* which allocation sites can query by matching predictor-sets; it can be refined at any time as further programs are analysed. Such queries can be built into the compiler or performed off-line and stored as (*allocation-site,lifetime*) pairs.

The use of confidence levels in the context of the experiments is reviewed below.

3.8.4 Using The Rules

In this thesis, the confidence level is used to define the conservatism of the prediction system. The higher the confidence level, the more likely the rule is to make a correct prediction (although it might be desirable to use a lower level to capture more predictions). Therefore, the level of conservatism can be varied by not considering rules with a low confidence level, in order to reduce the number of wrong pretenuring decisions made.

The reader is reminded that it is not essential to predict the lifetime of all objects. Rather, for objects that can be predicted, it is important to predict their lifetime with high confidence.

This is a crucial part of the system developed in this thesis, because varying the degree of conservatism of the system can have a big impact on performance. For instance, if pretenuring yields good performance for a certain program, it may be worth trying a more aggressive (lower) confidence level. Alternatively, if the pretenuring system yields bad results for a certain program, putting the confidence level at a higher level might improve performance.

The pretenuring scheme used works as follows:

1. Any rule with confidence less than a fixed threshold is exluded.

2. For any allocation site where the allocator and the allocatee match a set a

rules, the rule r with the highest confidence level was selected.

Theoretically, conflicting rules with identical confidences are possible. In practice, it was found that advice with a confidence higher than 80% never conflicts.

3.9 From Predictors To Allocation Sites

This chapter discussed the way data-mining rules are computed and associated with lifetimes to create a knowledge bank. Below is a description of the different ways by which the knowledge bank can be exploited.

3.9.1 Reusing Data-Mining Rules "As-Is"

While possessing data-mining rules is enough in theory to make predictions about the lifetime of an object, reusing it in the virtual machine "as is" would bear some significant disadvantages. For instance, a mechanism to discover the predictors associated with each object about to be allocated and match it with the knowledge bank would be required . This process would have to be performed on-line, typically by the class loader, as classes are loaded into the JVM.

The corollaries are as follows.

1. First, the class-loader would have to implement a potentially heavy infrastructure in order to discover the predictors (micro-patterns or software metrics) of each object about to be allocated.

2. Second, the compiler would have to determine (at run-time) the best allocation advice by matching the first mention of object predictors with the data-mining rules database.

3. Only then would the compiler be able to generate the allocation code (for the object to be allocated in the appropriate region).

This methodology has some obvious drawbacks. The infrastructure required to be created and inserted into various parts of Jikes RVM would be significant,

and the extra time-overhead induced by all the extra computation required would very unlikely be regained by the copying time saved during GCs.

3.9.2 Approach

To avoid a costly runtime overhead, off-line reusable advice files mapping allocation sites directly to lifetimes are computed. The format of these advice files is described in Section 4.2.

Considering a finite set of predictors $\mathcal{MP}$ and a lifetime $\mathcal{LT}$ (table 3), the knowledge bank initially contains mappings from predictors to lifetimes (mapping $\mathcal{PL}$). In order to prepare the advice files, the process below is followed:

1. All the allocation sites (*SiteID*) within the program to be executed are identified by parsing the program's bytecode and identifying every object allocation statement (= allocation sites).

2. For each allocation site, the source class and the destination class are located from the bytecode. Therefore, for each allocation site, is it possible to know the type of the object being allocated (destination) and the class that allocated the object (source).

3. Predictors of both source and destination classes are then identified by using the relevant predictor identifier tool. For instance, for this research on micro-patterns, a micro-patterns detector tool would be used, while for the research on software metrics, a metrics detector tool would be used. The mapping $\mathcal{S}$ is obtained.

4. Once the relevant predictors have been identified and associated with each allocation site, they are matched with the lifetime knowledge bank, and associated the relevant lifetime. If the knowledge bank does not contain advice for the predictors exhibited by a particular allocation site, then the site is considered "short-lived". The mapping $\mathcal{SL} = \mathcal{S} \circ \mathcal{PL}$ which associates sets of source predictors and destination predictors with a lifetime is produced.

5. At this stage, is it possible to associate a lifetime with every allocation site. Files matching allocation sites to lifetimes are then generated. The format of the advice files is described in Section 4.2.

Once these files have been generated, they are then ready to be used by the VM that can make use of the advices they contain. Chapter 4 discusses implementation details.

3.10 Testing Phase

The previous sections explained the way lifetime data was gathered, and the way lifetime predictions were generated. This section details the methodology used to test the predictions.

3.10.1 Test Benchmarks

As explained above, testing was performed on the jvm98 suite [147] which provides a set of well-tested benchmarks commonly used in the GC field to measure performance. jvm98 benchmarks are a set of benchmarks offering a well tested environment to run programs and allowing the comparison of findings with previous research. Also, because they run much faster than the *DaCapo* benchmarks, much more tests were performed than would otherwise have been possible, which helped refining the methodology.

3.10.2 A Generic Approach

Because training is performed on the *DaCapo* benchmarks and rules are applied to jvm98, the approach taken is generic: this is true prediction.

3.10.3 User Mode

To allow minimum interference from other applications of the system, all the tests were performed in single user mode[4] on this test Linux machine.

3.10.4 Best of Five Runs

When performing experiments, one could decide to run several iterations of the same program in the same JVM instance. At each iteration, objects predicted immortal would be allocated into the immortal space which is never collected. At the end of the iteration, all immortal objects allocated using the system would become garbage, reducing the amount of heap space available for the next iteration.

To avoid the above mentioned problem, each tests is run several times, in several different JVM instances. It was decided that each test would be run 5 times.

In the analyses, results from the test performed in the least wall-clock time are taken, since this is supposedly the test which encountered the least interference from other parts of the system. Also, the intuition behind taking the best of several runs is that the aim is to measure mostly program execution, and not other parts of the system such as class loading and JIT compilation. It is therefore assumed that this test has the most accurate value.

3.10.5 Compiler Replay Option

In a recent study, Blackburn et al. [25] advocate that due to non-deteminism in the JVM and the platform being tested, researchers should have a sound methodology to evaluate the performance of its optimisations. Although this paper had not yet been published at the time the experiments in this thesis were performed, the experimental methodology used matches their advice. Because tests are performed using an adaptive optimising compiler, doing measurements on the system

[4]Single user mode ensures that only the very necessary programs are loaded to allow the operating system to run.

is hazardous because the optimising compiler might not recompile methods at the same time from one run to another (see Section 3.3.3).

Jikes RVM provides a compiler replay option, allowing reuse of the profiling of the adaptive compiler from a previous run to allow fair comparisons. It ensures each method will be optimized at the same level from one run to another of the program in Jikes RVM, therefore removing the unpredictability of the adaptive compiler system, while keeping the same level of performance. In the experiments, this feature is used to improve the stability of results.

3.10.6 Varying Prediction Accuracy

The datamining tool associates a level of confidence to each predicted rule. Tests are performed by varying the minimum confidence level required for a rule to be taken into account.

3.10.7 Varying Heap Sizes

The impact of heap sizes on performance was also measured by performing tests on different heap sizes. The minimum heap size[5] required was experimentally determined by the system equipped with a copy mature-space for each *SpecJVM98* benchmark. Table 4 details the volume of data allocated, the minimum heap size and the maximum livesize associated with each benchmark.

The heap size is varied from the minimum heap size to 4 times the minimum heap size, so measurements are performed with 7 different heap sizes. Measurements are performed using the following multipliers of minimum heap size: 1, $1\frac{1}{3}$, $1\frac{2}{3}$, 2, $2\frac{1}{2}$, 3 and 4.

[5]The minimum heap size is the minimum heap size value necessary to run a program. Since a copy mature space is used, space for the copy-reserve also needs to be reserved.

Program	*Max. live (MB)*	*Allocated (MB)*	*Minimum heap size*
_201_compress	15 MB	133 MB	21 MB
_202_jess	12 MB	393 MB	22 MB
_205_raytrace	16 MB	255 MB	30 MB
_209_db	21 MB	144 MB	39 MB
_213_javac	21 MB	295 MB	40 MB
_222_mpegaudio	11 MB	54 MB	18 MB
_227_mtrt	21 MB	261 MB	38 MB
_228_jack	11 MB	408 MB	22 MB

Table 4: "Spec JVM98" benchmarks characteristics.

3.10.8 Programs Inputs

The DaCapo benchmarks provide three input sizes which are *small*, *default* and *large*. The DaCapo benchmarks were profiled using the small and large inputs to ensure the system is robust against most input sizes. However, input *default* was not profiled for any of the benchmarks, as this input is in some cases the same as input *large*, and including it may skew the training set.

The jvm98 benchmarks allow three input sizes which are *Speed 1*, *Speed 10* and *Speed 100*. Speed 1 is intended as a quick checkout of the benchmark programs on a JVM, while Speed 10 is provided for test purposes. Results are reported using input size 100.

3.10.9 Compiler Configuration

Jikes RVM allows many different build configurations. Researchers can configure the compilation level of the main system, as well as the JIT compiler to be used.

To measure the performance improvements of this method, tests are performed using a FastAdaptive configuration. This configuration means that the optimising compiler is used at both build-time and run-time.

3.10.10 Garbage Collector

Tests were run using an Appel-style generational collector[3]. In this scheme, objects are allocated in the nursery, and when the nursery is full, a nursery collection is triggered. The surviving objects are then copied into the mature space, increasing the size of the mature-space and decreasing the space available for the nursery accordingly. Consequently, the more space gets used by the mature-space, the less space is available for the nursery, and minor-collections become more frequent. Finally, when the size of the nursery is less than a certain threashold, a full heap collection is triggered to free some memory from the mature-space.

Although the default collector in Jikes RVM is currently a generational collector with a mature space managed by a *mark and sweep* policy (*genMS*), it was decided to use an Appel-style collector (*genCopy*) because this is the collector used by Blackburn et al. [26, 22] in their experiments. Since Blackburn et al.'s methodology is the best case scenario, the aim was to obtain results that are comparable to that point of reference.

Chapter 4

Implementation

Chapter 3 described the methodology employed to perform the experiments. The framework and the different steps involved in generating lifetime predictions were discussed. The framework is flexible and extensible to other measures than software metrics and micro-patterns.

In order to make use of these predictions, a system in the Jikes RVM Java virtual machine[1] which allocates objects in relevant parts of the heap according to their expected lifetime was developed. This chapter details the main characteristics of the system.

Section 4.1 gives an overview of the implementation. A description of the advice file format (see Section 4.2) is discussed and the way advice (see Section 4.3) is stored described. The allocation policy is discussed in Section 4.4. Finally Section 4.5 reviews the performance overhead induced by the implementation, and Section 4.6, to create a point of reference, details the performance gains of the system when doing self-prediction.

4.1 Implementation Overview

Chapter 3 described the experimental platform used for this research. This section provides a brief overview of the way the advice system was implemented into Jikes RVM. This section is written as a way to give the reader a general view of the

system, as opposed to a fine-grained understanding of the system. The following sections of this chapter, on the other hand, explain the system in more detail.

4.1.1 Command-line Parameters

To perform allocation decisions, the virtual machine needs to query the pre-computed knowledge bank. There are several ways this mechanism could be implemented.

One strategy would be to hard-code the knowledge bank into the JVM, and recompile it. This strategy would avoid having to dynamically load the knowledge bank at each JVM execution, but would lack flexibility, since any changes in the knowledge bank would require the JVM to be recompiled.

A second strategy would involve storing the knowledge bank in separate files that would be loaded and burned into Jikes RVM's allocator at the time the JVM is being compiled. This strategy is more flexible than the previous one since no code needs to be rewritten when upgrading the knowledge bank. Unfortunately, this technique still requires the JVM to be recompiled every time the policy is changed.

Because it is not desirable to have to recompile the JVM every time the knowledge bank is upgraded, it was decided to go for a third strategy. Advice is kept in unique program-specific files which are derived from the knowledge bank. They are computed ahead of time based on the predictors exhibited by each program. The name of the advice file to be used is supplied to the JVM using command-line arguments.

A high-level explanation of the way the JVM is initialised is as follows. Initially, the JVM is started by the boot program, which is responsible for doing the low-level preparations such as establishing the initial virtual memory map and installing the C wrapper functions in memory so that they are accessible by the JVM. Once this step is complete, all further initialisation of the JVM is done in Java, or by using the previously registered C wrapper functions. Initialisation continues by performing a variety of operations such as initialising the memory

manager, the JIT compiler and the class-loader. Finally, the JVM parses the command line arguments and loads the input file provided, before loading and executing the program class supplied on the command line.

By using this strategy, new allocation policies can be explored without the need to recompile the Jikes RVM.

4.1.2 Input File

When the input file is loaded into the JVM, it is then parsed and allocation advice is stored into an internal structure. For more information regarding the advice file format, refer to Section 4.2. For more information regarding the way advice is stored internally, refer to Section 4.3.

4.1.3 Allocation Policy

Once the advice file has been loaded and parsed, an internal structure holds information mapping allocation sites to lifetimes. When a site is compiled by the JIT compiler, the system checks if any advice for the allocation site is available, and if so, that advice is burned into the generated code as a constant. If no information is available for a particular allocation site, the default advice is inserted into the generated code. Therefore, the system only needs to check for advice when a site is compiled, not at every object allocation.

Under the default policy of the system, large objects are allocated in a special region called *Large Object Space* (LOS) where they are managed separately. Some VM specific objects required by the JVM are allocated in the *immortal space*. All other objects are allocated into the nursery.

More information regarding the allocation policy, is given in Section 4.4.

4.2 Advice Files Format

Section 3.9 described how advice files are created for each particular program about to be run. This section describes the format used to match allocation sites with advice.

In Java, every class and interface has a fully qualified name used to uniquely identify it. For example, the fully qualified name of the class "String" in Java is "java.lang.String".

The Advice files contain a line for each allocation site whose lifetime can be predicted. Each line displays the fully qualified name of the class in which the allocation occurs, followed by the method in which the allocation occurs, followed by its offset in the bytecode and the advice. This format can be expressed as follows:

ClassName:MethodName:Offset prediction

"ClassName" is the fully qualified name of a class, "MethodName" is the fully qualified name of the method, and "Offset" is the bytecode offset of the allocation statement within the method. For example:
Ljava/security/AllPermission;:newPermissionCollection()Ljava/security/PermissionCollection;:0 2

The fully qualified name here shows a call site from the class *AllPermission* (from package *java.security*), in method *newPermissionCollection()*, which returns a *PermissionCollection* object, and the offset within the method is 0. Regarding the lifetime convention, 0 means *short-lived*, 1 means *long-lived* and 2 means *immortal*. Consequently, the advice predicts that this allocation site allocates mostly immortal objects.

4.3 Storing Advice In Memory

The previous section showed how advice files are created. This section analyses how advice is stored in Jikes RVM's memory.

4.3.1 Parsing The Input File

When reading an advice file, the aim is to extract its advice and store it into memory to be reused at a later point in the program.

To this end, Chris Ryder, at the University of Kent added a parser to the JVM. For each line of the input file, the parser extracts the allocation site along with the allocation advice associated with it. Mappings between allocation sites and allocation advice are then stored into memory in one of two ways, discussed below.

4.3.2 Using a HashMap

The storage mechanism used to store advice should require as little memory as possible while allowing very fast retrieval of data. In his first prototype, Ryder stored information in a "HashMap", mapping fully qualified names with lifetime. For instance, the storing mechanism associated a key object (String class, String method, String type, int offset) with its lifetime (int):

(String class, String method, String type, int offset) *key* $\rightarrow$ (int lifetime) *value*

In the association above, *class* is the fully qualified name of the class in which the allocation site resides. Likewise, *method* is the name of the method containing the allocation site, while *type* is the type returned by the method and *offset*, is the offset at which the allocation site can be found.

The way this storage mechanism can be improved is investigated below.

4.3.3 Improving The Storage Mechanism

The naive implementation used in the first prototype suffers the overhead of having to store fully qualified name strings for each allocation site possessing advice, therefore incurring duplication of information since none of those strings are shared. For example, all the sites of a single class would duplicate the *class* String.

Ryder therefore developed a less memory-intensive approach. JikesRVM's class loader maintains its own HashMap holding strings referring to class names, method names and type names, and ensures there are no duplicates. In the class loader, these Strings are wrapped into objects called *VM_Atom*, which are used to represent names, descriptors, and string literals appearing in a class's constant pool.

Rather than associating raw Strings with advice, Ryder now reuses the class loader's information and maintains the HashMap as follows:

> (VM_Atom class, VM_Atom method, VM_Atom type, int offset) *key*
> → (int lifetime) *value*

Note that the *VM_Atom*s are now just references to the class loader's *VM_Atom*s. As a consequence, reusing *VM_Atom*s that would have been allocated anyway can reduce the memory requirements of the system.

4.4 Allocation Policies

As described in Section 4.2, the format of the advice files offer an estimated prediction of the lifetime of objects allocated at specific sites. However, it is the responsibility of the JVM to decide how to use this advice and in which region to allocate an object. This section explains the allocation policies used during the experiments.

In the Jikes RVM, every new object is allocated via the same allocation method, which takes in a parameter known as the *allocator*. The allocator defines the space in which the JVM would normally allocate an object. For instance, in

the case of a generational collector with a copy mature space, the allocator can be *nursery*, *mature space*, *large object space* or *immortal space*.

Ryder modified the compiler and the allocation method so that for every new object, an *advice* parameter is passed to the method, along with the default *allocator* parameter given by the JVM. This way, the allocation method is supplemented with a custom advice.

When an allocation site is being compiled by the JIT compiler, the system checks if any advice for the allocation site is available. If so, that advice is burned into the generated code as a constant.

The allocation method then has to decide the space in which the object needs to be allocated, based on the allocator and the advice (see following Section).

4.4.1 Whom To Trust?

For each object allocated, it is necessary to make the following choice: "Should the advice system or the allocator provided by the JVM be trusted, given the JVM might happen to know more about this particular object?". Unfortunately, there is no definite answer to this question, and different cases require different decisions. For instance, if the JVM's allocator is the default space, it might be assumed that the JVM does not know what to do with this particular object. On the other hand, if the JVM's allocator is the immortal space or the large object space, then following the allocator is probably a good idea.

It was decided to follow only the advice when the allocator suggests allocating in the default space. When the allocator wishes to allocate in any other region, it is assumed that the JVM knows more about the object than the advice does, and therefore follow its policy.

By default, Jikes RVM allocates all objects of certain VM packages in the immortal region, because these are not expected to die before the end of the program. Examples of such packages include GCSpy "Lorg/mmtk/vm/gcspy/" a heap visualisation framework [127], and Jikes RVM's memory management subsystem "Lorg/mmtk/".

Advice / *Allocator*	*Short-lived/No Advice*	*Long-lived*	*Immortal*
Nursery	Nursery	Mature	Immortal
Mature	Mature	Mature	Immortal
LO Space	LO Space	LO Space	Immortal
Immortal	Immortal	Immortal	Immortal

Table 5: Allocation Policy
LO Space: large object space.

Table 5 summarizes this allocation policy.

4.4.2 Implementing The Switch

For every new object allocated, the allocation method has to decide whether to trust the advice or the allocator. Implementing a combination of *if statements* that would be executed at every new allocation could prove costly.

In the implementation, binary values were associated to each allocator, each advice and each space. Binary values were chosen in such a manner that the result of a *binary AND* between the allocator and the advice would give us the code of the space in which to allocate the object. This way, only a single bytecode instruction is required to decide in which space the object should be allocated.

In the implementation of Jikes RVM modified for this thesis, the *nursery* allocator is given a value of 0xF (1111), the *mature space* allocator a value of 0xC (1100), the *large object space* allocator a value of 0x9 (1001), and the *immortal space* allocator a value of 0x0 (0000). Similarly, *short-lived* advice is given the value 0xF (1111), *long-lived* advice is given the value 0xC (1100), and the *immortal* advice is given the value 0x0 (0000). Table 6 summarises these values and shows how they are used.

Advice / *Allocator*	*Short/NA(1111)*	*Long(1100)*	*Imm(0000)*
Nurs (1111)	Nurs (1111)	Mat (1100)	Imm (0000)
Mat (1100)	Mat (1100)	Mat (1100)	Imm (0000)
LOS (1001)	LOS (1001)	LOS (1001)	Imm (0000)
Imm (0000)	Imm (0000)	Imm (0000)	Imm (0000)

Table 6: Advices And Spaces Binary Values

4.5 Advice Loading Overhead

This section investigates the execution time overhead incurred by loading and parsing the advice file.

All the tests are performed using a generational copy collector memory management system. Jikes RVM is compiled using the *FastAdaptive* optimising compiler (see Section 3.3.3). In this configuration, the compiler turns off all assertion checks and uses an adaptive JIT compiler capable of optimising frequently used methods. Details of the experimental platform are available in Section 3.2. For a complete description of the experimental methodology, please refer to Chapter 3.

Loading an advice file and storing its information has a cost, both in terms of memory space and loading time. These overheads are discussed in the following sections.

4.5.1 Execution Time Overhead

To estimate the execution time overhead, the regular execution time of the VM running a test benchmark is measured against the execution time of the VM running the same benchmark, which also loads a neutral advice file where all advice is short-lived. Unlike regular performance tests, in which the interest mostly lies in GC time, the focus is here on the execution time of the entire program (using the *time* command on linux), including the VM bootup time. The reason for this is that the overhead of loading the file is carried by the virtual machine itself at bootup, so the best way to understand the global impact that loading an advice

file has on the system consists on timing the execution time of the entire VM using an external tool. Also, Jikes RVM's time statistics were not used for these experiments because Jikes RVM cannot start gathering any time statistics before the entire JVM is fully booted, which is precisely the aim of this experiment.

With the exception of using the *time* command instead of Jikes RVM's reportable time statistics, the methodology employed to calculate the time overhead of the system is the same as described in Chapter 3. In particular, the optimising compiler settings, the compiler replay options, the "best of 5 runs" approach, the varying heap size, and the garbage collector used are the same as described in Chapter 3.

The Tests

The implementation was tested using SPEC's jvm98 benchmarks with different heap sizes, shown as multiples of the minimum heap size required to successfully run the program. Note that Jikes RVM allows the heap size to be dynamically resized using some heuristics to determine when a bigger or smaller heap would be more efficient. Since this can have unexpected impact on the performance tests, it was decided to fix the minimum heap size and the maximum heap size to the same value to ensure this does not happen.

jvm98 allows benchmarks to be run using three different problem sizes, namely *Speed 1*, *Speed 10* and *Speed 100*. These tests follow SPEC's recommendation that reportable results must be run with a problem size of 100.

Figure 4 shows the relative degradation or improvement of the program's runtime with a neutral advice file against the default run. Here, *+2* would mean that loading the advice slows down the program by an average of 2%.

For clarity purposes, these results are also reported in Table 7. In this table, each line corresponds to a specific heap size, and each column to a specific benchmark.

This data show that the advice system induces a small overhead in execution

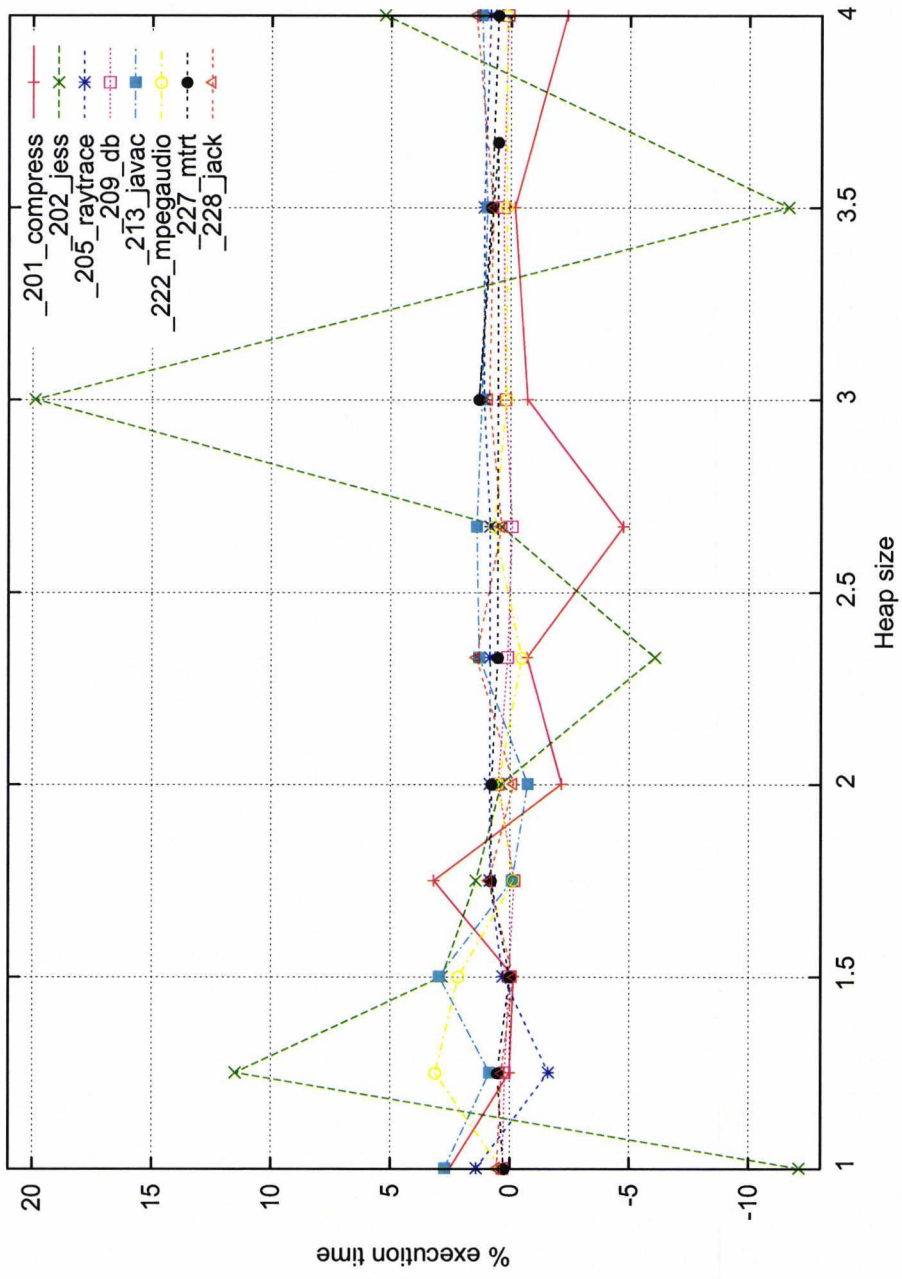


Figure 4: Advice system time overhead, *Speed 100*

Bench	*201*	*202*	*205*	*209*	*213*	*222*	*227*	*228*	*Avg*
Heap size									
1 X	+2.49	-14.66	+1.38	+0.28	+2.70	+0.33	+0.26	+0.52	-0.84
1.25 X	+0.00	+17.23	-1.64	+0.19	+0.82	+3.11	+0.52	+0.34	+2.57
1.5 X	-0.17	+2.82	+0.27	+0.00	+2.94	+2.15	+0.00	-0.17	+0.98
1.75 X	+3.17	+1.41	+0.82	-0.19	-0.11	-0.16	+0.79	+0.87	+0.82
2 X	-2.18	+0.35	+0.83	+0.47	-0.76	+0.49	+0.78	-0.17	-0.02
2.33 X	-0.72	-6.09	+0.82	+0.09	+1.28	-0.49	+0.52	+1.40	-0.40
2.67 X	-4.75	+0.35	+0.82	-0.09	+1.39	+0.66	+0.52	+0.35	-0.09
3 X	-0.73	+19.86	+1.09	+0.19	+1.18	+0.16	+1.32	+0.88	+2.99
3.5 X	-0.19	-15.54	+1.09	+0.28	+0.96	+0.16	+0.79	+0.71	-1.47
4 X	-2.40	+0.29	+0.82	+0.09	+1.18	+0.16	+0.52	+1.43	+0.26
Average	-0.55	+0.60	+0.63	+0.13	+1.16	+0.66	+0.60	+0.61	+0.48

Table 7: Advice file loading time percentage overhead over benchmarks' default execution times.

time.

Because the maximum heap size is fixed and the advice system is stored into the default Jikes RVM immortal space, the space devoted to the nursery and the mature space is reduced. This may affect GC scheduling and hence increase the amount of live data that needs to be promoted at the first GC. This effect may also accumulate with further GCs.

_202_jess in more details

Table 7 shows that *_202_jess* has a rather erratic behaviour, with spikes of up to 20% overhead. The analysis below shows the reasons for this behaviour. Figure 5 shows the absolute execution times for both *_202_jess* not loading an advice file, and *_202_jess* loading a neutral advice file.

For clarity purposes, this paragraph uses the notation 'A' to refer to the case when an advice file is loaded, and 'NA' to refer to the case when no advice file is loaded.

Unexpected behaviours happen at heap sizes 1X, 1.5X, 3X, 3.5X, and 4X. A big jump from 1X to 1.5X is noticed. The height of the jump can be explained by the fact that between 1X and 1.5X, where NA performs badly, A performs better, and vice-versa. Likewise, at 3X and 4X, A performs worse than NA, whereas at 3.5X, NA performs worse than A.

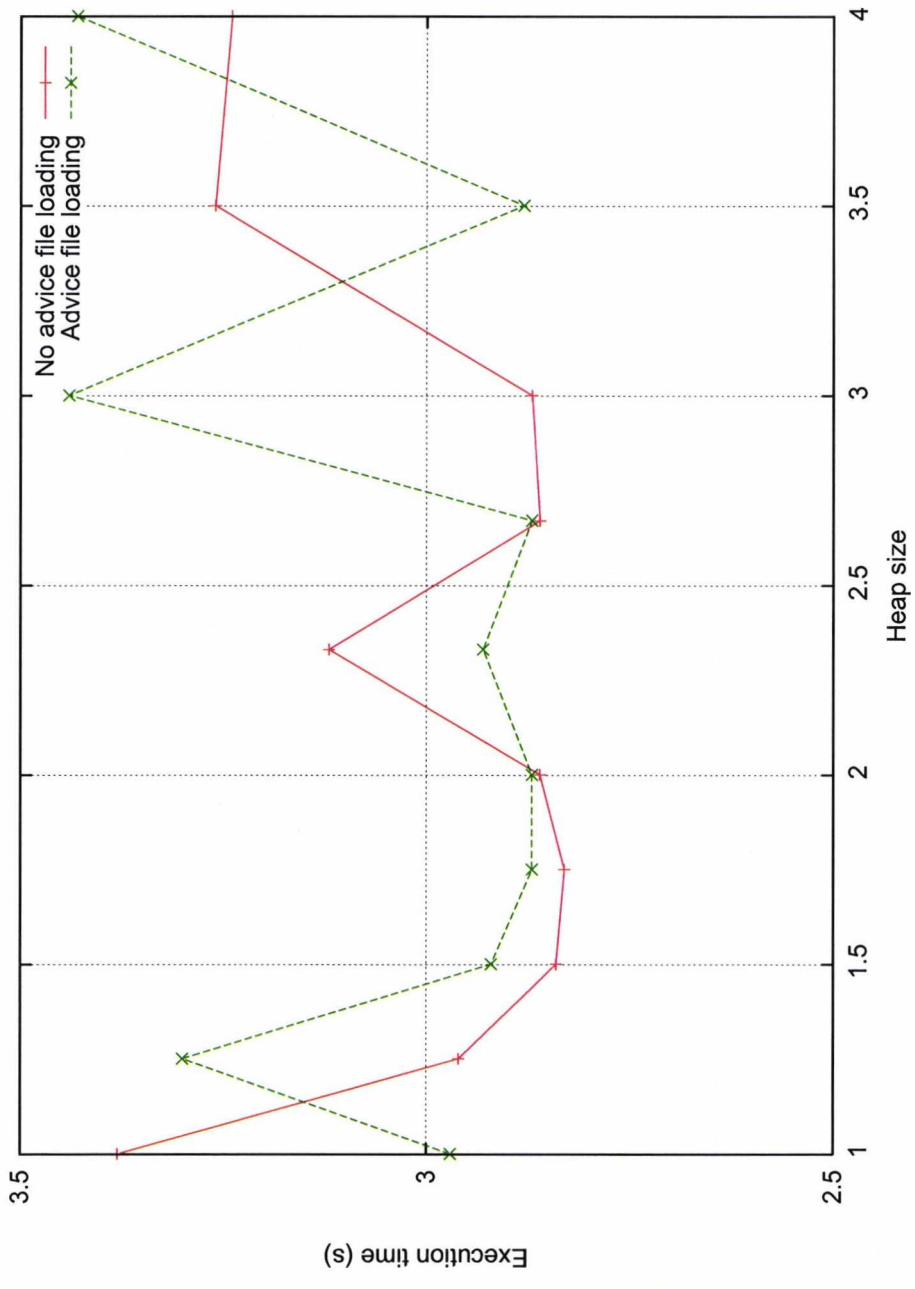


Figure 5: Advice system time overhead for '*_202_jess*', *Speed 100.*

Heap size	No advice loading (Minor, Major)	Advice loading (Minor, Major)
1 X	298,1	304,1
1.25 X	88,0	93,0
1.5 X	43,0	44,0
1.75 X	33,0	33,0
2 X	27,0	27,0
2.33 X	21,0	21,0
2.67 X	18,0	18,0
3 X	16,0	16,0
3.5 X	14,0	14,0
4 X	13,0	13,0
5 X	13,0	13,0

Table 8: Garbage collection counts for '*_202_jess*', *Speed 100.*

In general, due to non-determinism in the JVM, both A and NA are unstable in terms of execution time as the heap grows bigger, which explains the variations in relative execution time.

Next, *_202_jess* benchmark is analysed in Table 8 which shows the minor garbage collections, and full garbage collections triggered during the run of the program. Each column shows two numbers separated by a comma. The first number indicates the total number of collections (both minor and major) that occured during the run of the benchmark, while the second number indicates the number of major collections that occured.

At minimum heap size, 1 full heap collection had to be triggered in both cases. However, as the heap grows bigger, minor collections alone were sufficient to reclaim space for the application to run.

At tight heaps (1 X to 1.5 X), the system (loading an advice file) requires more minor GCs to happen.

An average percentage overhead of 0.48% is reported. In real life, this overhead is likely to be even less significant since programs would typically run much longer than the *SpecJVM98* test benchmarks.

4.5.2 Estimated Space Overhead

To estimate the space overhead of the advice system, code was inserted inside Jikes RVM to print out the heap usage of the system. Loading the advice file involves some extra work to be performed by the JVM, such as loading Java classes like *HashMap* into memory earlier than it would otherwise. Likewise, some *VM_Atoms* have to be created. For this reason, it was decided to compare heap usage at two key points: before the program starts running and at the end of the program. The intuition is that at the end of the program, these classes that were loaded earlier due to the modifications made for this work would have been loaded anyway.

At each of these key points, the heap usage of the system is compared with a neutral advice file, against the heap usage of the system without such advice. To ensure fair comparison, a full heap GC is triggered before each of the key points to make sure the heap does not contain any garbage.

Table 9 below summarises the findings. The first part of the table shows the results for the first key point, and the second part of the table the results for the second key point. Note that apart from the number of advice lines in the advice file loaded (*Nb Ad*), all the data is expressed in numbers of pages, each page having a size of 4KB. This is because Jikes RVM only keeps count of how many pages are being used at any given time by each region. Hence there is an error margin of plus or minus 4KB.

At key point 1, the system constantly uses 8 pages more in the immortal region when loading advice. In one case however ("_213_javac"), the system also require 8 extra pages in the mature space.

At key point 2, a constant increase of 8 pages is witnessed in the immortal space when loading an advice file. In 2 cases ("_209_db" and "_213_javac"), 8 extra pages are also used in the mature space.

In most cases, as expected, the overhead at the first key point is higher than the overhead at the second keypoint. This is due to some *VM_Atoms* being generated earlier than they would otherwise when loading an advice file, which

		Benchmark	201	202	205	209	213	222	227	228
		Nb Ad	363	523	463	373	656	426	462	538
Key point 1 (before)	No advice	Boot	0	0	0	0	0	0	0	0
		Immo	32	32	32	32	32	32	32	32
		Meta	1	1	1	1	1	1	1	1
		LOS	884	884	884	884	884	884	884	884
		Nurs	0	0	0	0	0	0	0	0
		Mature	1024	1024	1024	1024	1024	1024	1024	1024
		Total	1941	1941	1941	1941	1941	1941	1941	1941
	Advice	Boot	0	0	0	0	0	0	0	0
		Immo	40	40	40	40	40	40	40	40
		Meta	1	1	1	1	1	1	1	1
		LOS	884	884	884	884	884	884	884	884
		Nurs	0	0	0	0	0	0	0	0
		Mature	1024	1024	1024	1024	1032	1024	1024	1024
		Total	1949	1949	1949	1949	1957	1949	1949	1949
		Overhead	8	8	8	8	16	8	8	8
Key point 2 (after)	No advice	Boot	0	0	0	0	0	0	0	0
		Immo	32	40	32	32	40	32	32	32
		Meta	4	2	2	3	4	2	2	2
		LOS	1026	1161	1156	1026	1179	1159	1156	1178
		Nurs	0	0	0	0	0	0	0	0
		Mature	1136	1320	1184	1128	1600	1248	1184	1368
		Total	2198	2523	2374	2189	2823	2441	2374	2588
	Advice	Boot	0	0	0	0	0	0	0	0
		Immo	40	48	40	40	48	40	40	40
		Meta	4	2	2	3	4	2	2	2
		LOS	1026	1161	1156	1026	1176	1159	1156	1178
		Nurs	0	0	0	0	0	0	0	0
		Mature	1136	1320	1184	1136	1608	1248	1184	1376
		Total	2206	2531	2382	2205	2836	2449	2382	2604
		Overhead	8	8	8	16	13	8	8	16

Table 9: Advice file space overhead over benchmark default space usage.

are then shared with the compiler. The biggest space overhead observed is 16 pages (64KB), although most of the time, the overhead is 8 pages (32KB). Note that a tuple is used in order to record the number of pages in each region, and this mechanism has itself some space overhead which is reflected in table 9.

These figures show that the space overhead of the system is minimal.

4.6 Self Prediction Performance

This section analyses the performance of the implementation using self-prediction. The trace files previously generated are analysed and their lifetimes are computed using Blackburn et al's formula (refer to Section 2.4.7 for more details). Because advice is generated based on past runs of the program, self-prediction provides each allocation site of each program the best possible lifetime advice.

By applying Blackburn et al's formula (see Section 2.4.7), one advice file per benchmark is generated, matching each allocation site with its associated lifetime. For each experiment, Jikes RVM is given the advice file corresponding to the program about to be run.

4.6.1 GC Time Performance

Figure 6 presents a performance graph (as percentage improvement in GC time) of the system using self-prediction. This set of experiments was performed using the homogeneity thresholds described in Section 3.6.1. The methodology employed to carry out these experiments is the same methodology as described in Section 3.10.

This graph shows constant improvements in GC time of up to 80% for most programs.

However, '_228_jack' behaves badly as the heap grows bigger. Table 10 shows the number of GCs triggered during a run of the program, both with and without advice, at each specific heap size. As the heap becomes large (> 3 times), the number of GCs with and without advice remains constant at 16 minor GCs.

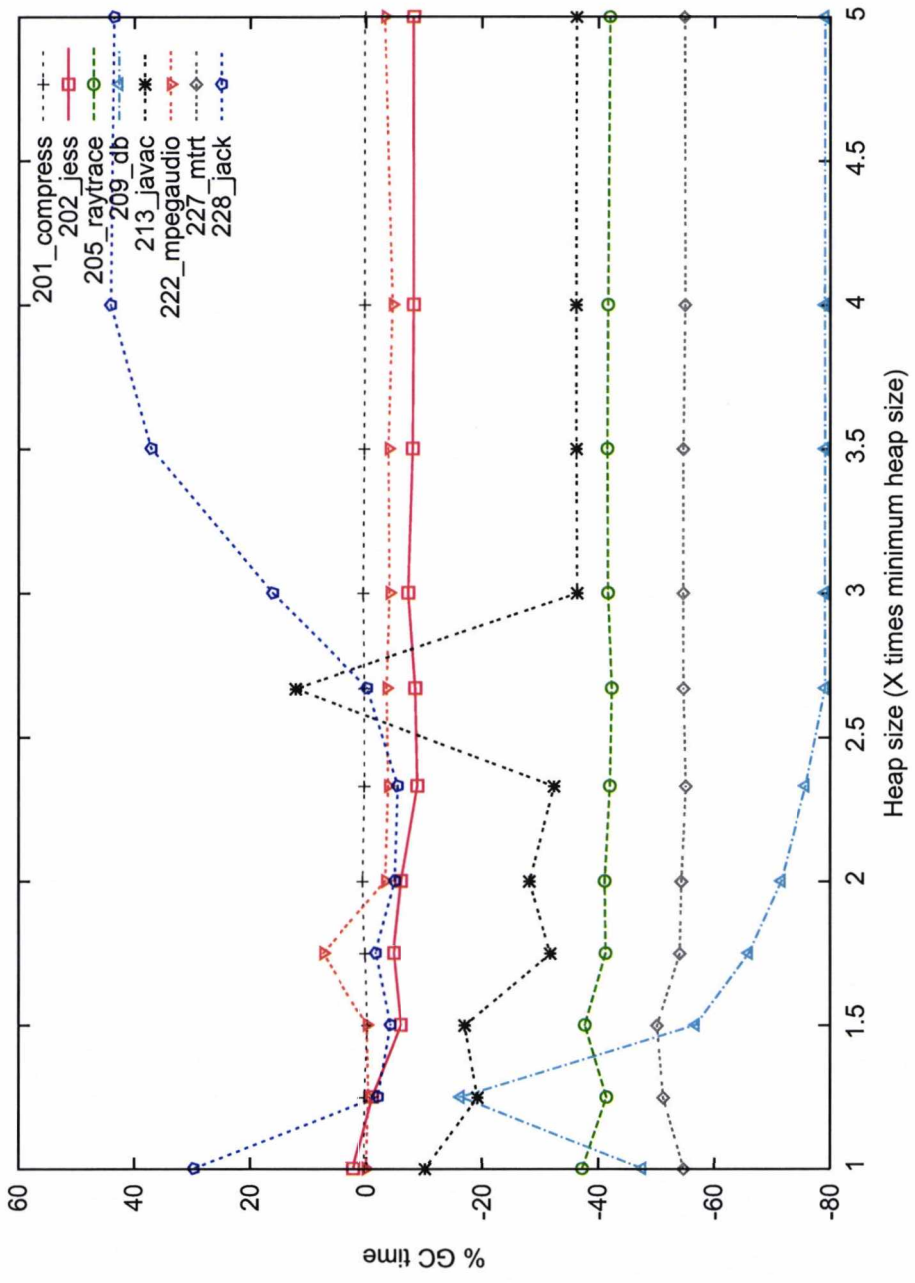


Figure 6: Self-prediction: improvement in GC time, *Speed 100.*

Heap size	No advice loading (Minor, Major)	Advice loading (Minor, Major)
1 X	115,2	149,3
1.25 X	104,1	129,2
1.5 X	37,1	38,1
1.75 X	102,1	124,1
2 X	57,0	59,0
2.33 X	28,0	29,0
2.67 X	22,0	22,0
3 X	18,0	18,0
3.5 X	16,0	16,0
4 X	16,0	16,0
5 X	16,0	16,0

Table 10: Garbage collection counts for '*_228_jack*', *Speed 100.*

It is reasoned that when using the advice, since pretenuring is performed with a fixed sized heap, less space is available for the nursery. Therefore, GCs are triggered earlier, and objects are given less time to die. The expected consequence is that more objects survive and need to be copied into the mature space, hence increasing GC time. Note that this was not noticed by Blackburn et al. in their study, because they do not consider heap sizes large than three times the maximum heap size.

This experiment confirms Blackburn et al.'s findings that pretenuring can offer improvements in GC time (of up to 80%). However, making wrong predictions can sometimes harm the program and predictions must therefore be used with caution when deciding on the pretenuring policies.

4.6.2 Performance Over Execution Time

It was shown that improvements over GC time can be high using self-predicted pretenuring (the best possible scenario). Unfortunately, GC time improvements do not always translate into throughput improvements. This section discusses the reasons for this.

Reference fields from immortal objects need to be remembered in order to

avoid them being garbage collected at the next GC. In MMTk, this is achieved by using the remembered set during minor collections. However, at major GCs, it is achieved by scanning the immortal space to ensure that only reachable immortal objects are traced.

The downside of this technique is that immortal objects need an extra mark-bit to be set during the marking phase. Every time a new immortal object is allocated, the allocator needs to set the mark-bit to the appropriate value, requiring extra computation and reducing throughput. On the other hand, by considering only reachable immortal objects as part of the root set, this technique reduces nepotism (see Section 2.4).

Furthermore, objects consume less space in the immortal region than they would in the mature space because there is no need to keep a copy reserve. When pretenuring into the mature space, objects consume twice as much space as they would in the immortal space because of the necessity to keep a copy-reserve. This reduces the space available for the nursery, and can increase the number of GCs required. Furthermore, the more pretenuring is employed, the greater the chance the write barrier slow path[1] has to be taken, slowing down the overall throughput.

The overall throughput of the system is studied using self-predictions as opposed to the default case.

Graph 7 details the relative throughput improvement using self-prediction at input size 100. On average, throughput is improved by up to 20% (*_202_jess*). However, some benchmarks, such as *_209_db* show degradations of up to 9%.

"_209_db" has the pecularity of being sensitive to the layout of data since it repeatedly traverses long singly-linked lists [22]. Also, since GC time only accounts for a small fraction of execution time (see Tables 8 and 9), the GC improvements introduced can easily be overridden by small mutator overheads. Studying the

[1] At every pointer write, the write barrier compares the address of the source pointer with the address of the destination pointer. If the source pointer is in the mature space or the immortal space and the destination pointer is in the nursery, then the source pointer needs to be recorded in a buffer, routinely refered to as *remset*. This operation of adding a reference inside the *remset* is performed by following the write barrier's *slow path*.

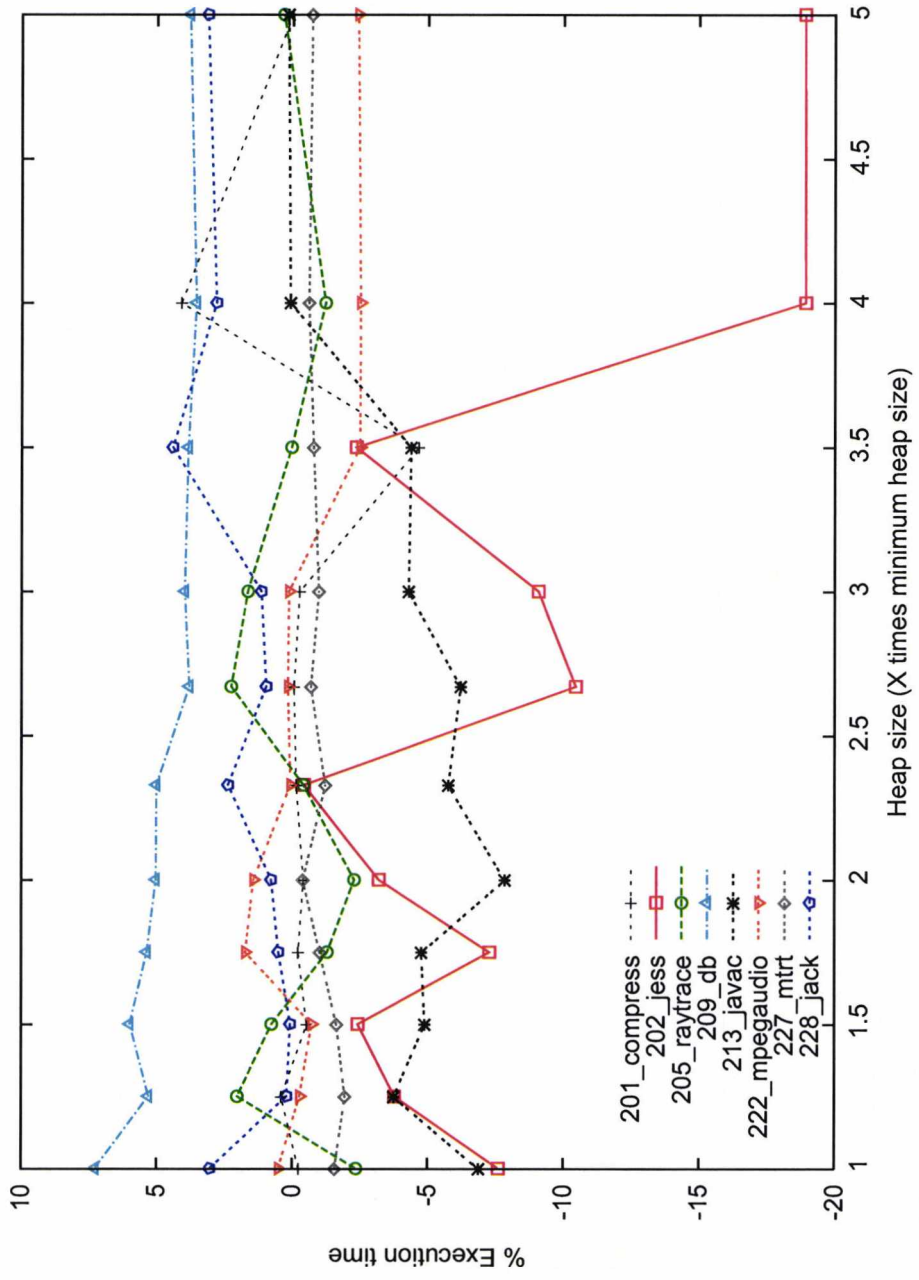


Figure 7: Self-prediction: throughput improvement, *Speed 100* (lower is better).

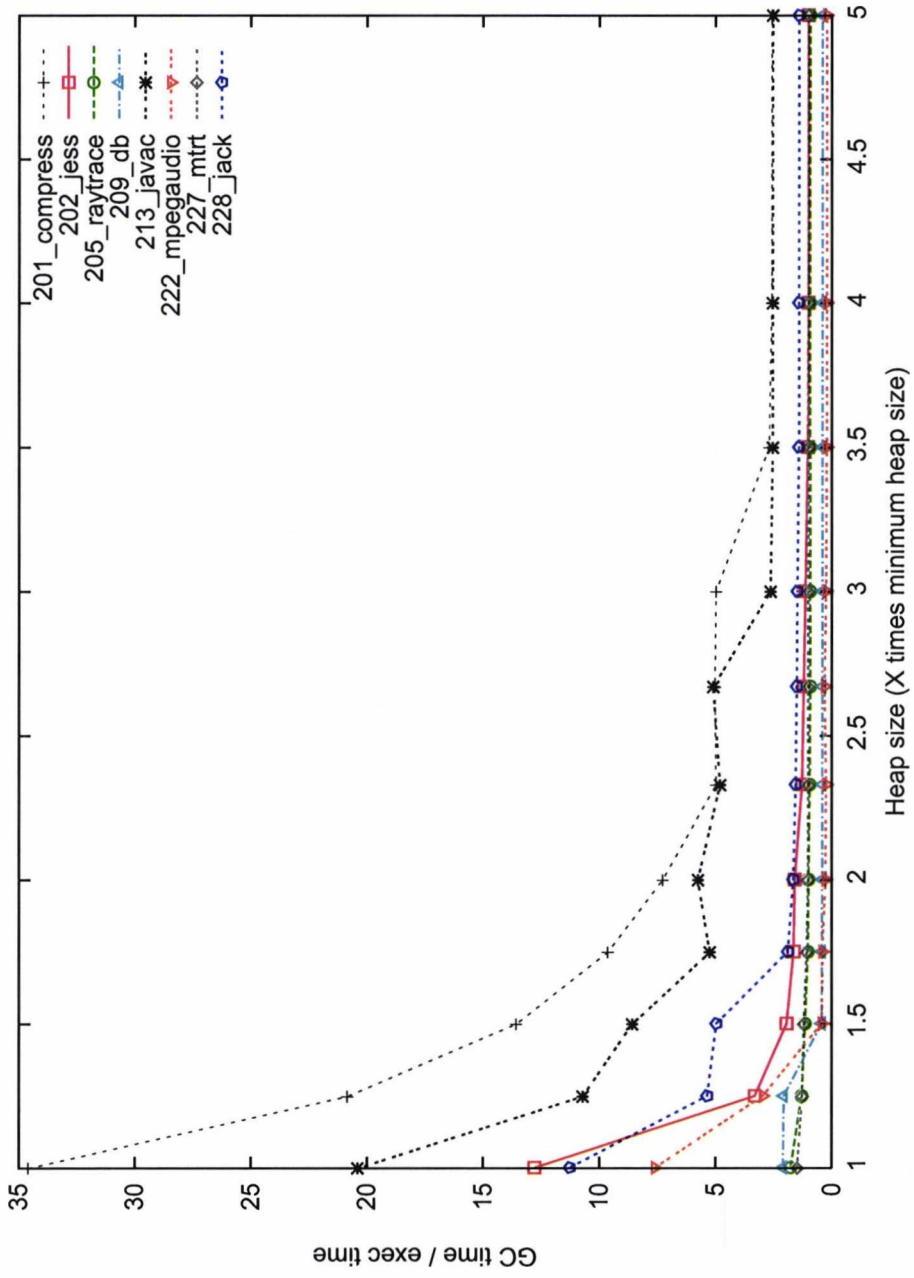


Figure 8: Self-prediction: GC time as a fraction of throughput, *Speed 100.*

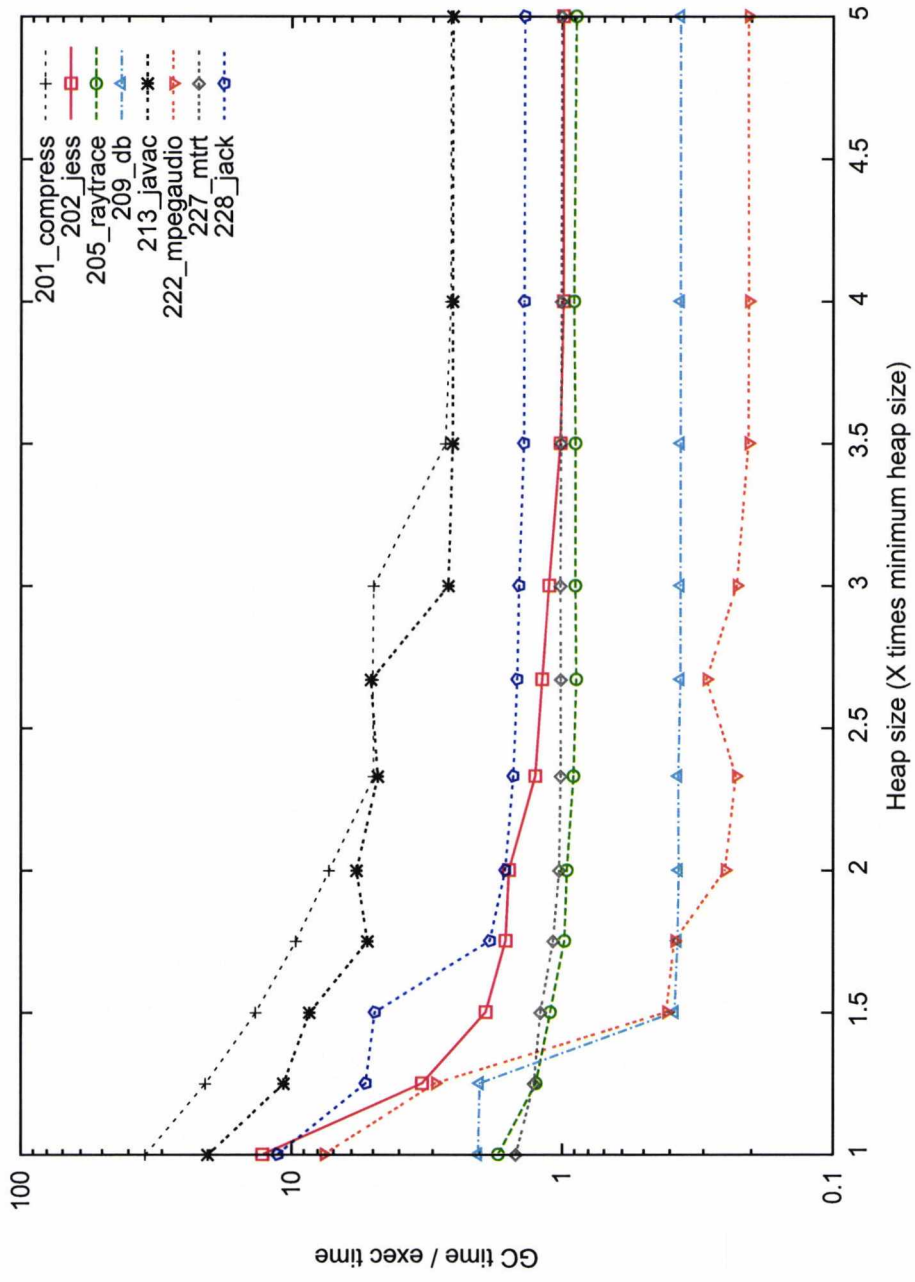


Figure 9: Self-prediction: GC time as a fraction of throughput, *Speed 100*, logarithmic scale.

hardware performance counters for _209_db revealed that despite a similar cache behaviour for the mutator, there is an increase in DTLB misses[2] of 33%.

This proves that even using the best predictions, pretenuring can, while diminishing the time spent doing GC, actually reduce a program's throughput. The following sections discuss these aspects further in the context of this work.

4.6.3 Self-Prediction Conclusion

Pretenuring has interesting potential when applying appropriate predictions. Improvements in GC time can be as great as 80%. However, even with the best predictions, pretenuring can in some cases impose an overhead on the overall program execution time.

4.7 Conclusion

This chapter showed how the system is implemented to take advantage of ahead-of-time lifetime predictions. As far as performance is concerned, this implementation has a low runtime overhead and the potential benefits can be significant (self-prediction). Chapter 6 shows how we can predict lifetime ahead of time using static properties captured by micro-patterns. Chapter 5 describes how is is possible to take advantage of more general program engineering properties to predict object lifetimes using software metrics.

[2]A Data Translation Look-aside Buffer (DTLB) is used to perform address translation from virtual addresses to physical page addresses (and vice-versa) for data.

Chapter 5

Using Software Metrics

The previous chapters of this thesis explored how it is possible to take advantage of object lifetime predictions. Previous trace based self-prediction techniques are extremely time consuming [39, 84, 26, 22]. This is because they require gathering and analysing gigabytes of data in order to generate lifetime predictions. Other dynamic techniques do not usually incur significant speedups [73, 96].

It was discussed in chapter 3 how the use of accurate *predictors* would allow the exploitation of lifetime predictions at a much cheaper analysis cost than is currently required [26, 22]. A simple static analysis of the program would allow us to identify *predictors* stored in the lifetime knowledge bank and make pretenuring decisions.

Chapter 3 discussed the framework developed for creating a knowledge bank matching specific predictors with lifetimes. Creating the knowledge bank involves gathering traces from a training set, computing the lifetimes of objects and allocations site, and matching this information with predictors. This information is then data mined, and the rules generated by the data mining software are stored in the knowledge bank. As mentioned in previous chapters, this thesis explores the use of two types of predictors: *Software Metrics* and *Micro-Patterns*.

This chapter reviews the use of *Software Metrics* as a predictor. Most of the work in this chapter was conducted in collaboration with Jeremy Singer from the University of Manchester.

The first Section of this chapter introduces the notion of *software metrics* and describes those used. The second Section of this chapter presents an information theoretic analysis of the software metrics selected. The third Section shows the results obtained using this approach in the experimental setup.

5.1 Software Metrics

5.1.1 What Are Software Metrics?

With the ever increasing complexity of computer systems, software architects need to design flexible structures and ensure the quality of the code developed. The emergence of Object Oriented (OO) programming and its advantages have helped software architects in several ways [119]. Systems became more robust and more reliable while code reuse saved a lot of development time. At the same time, however, the complexity of software systems started to increase dramatically [97]. Therefore, it became important for software architects and project managers to assess the quality of the code being developed. New focuses on software security make the quality of code an even more important concern.

In very large systems, however, assessing the quality of the code is an extremely challenging problem. Software metrics were developed in an attempt to produce a concise and quantitative analysis of certain aspects of a software system. They help to describe the quality and complexity of a software system in an impartial and objective way. They can be used to assess the overall quality of code from a particular programmer. Finally, they are easy to compute, and tools are usually readily available.

5.1.2 Which Software Metrics Are Available?

Over time, a number of software metrics have been proposed and used. All have a different focus and can be used to describe many aspects of a system. A few of these metrics are detailed below.

The easiest software metric to calculate is probably *Source Lines Of Code* which consists on counting the number of lines of code in a program and is commonly used to estimate the amount of effort that will be required to develop a program [15]. This metric, however, does not take into account the complexity of the code nor the interaction between different components of the program.

Cyclomatic Complexity [111] measures the complexity of a program by calculating the number of linearly independent paths within a program's source code. In essence, the more linearly independent paths there are, the more complex the program is. For the purpose of this work, however, it is not possible to derive object lifetimes from a measure of the complexity of a program itself. Rather, it would be interesting to measure the complexity of a *class*.

The *Function Point* [97] metric measures the usefulness of the program in terms of the functionality that is perceived by the user. Functions interacting with the user are split into five categories: outputs, inquiries, inputs, internal files, and external interfaces. These are assessed for complexity and awarded a number of function points. This measure is end-user focused and tends to underestimate the necessity of internal functions or algorithms. This metric is also difficult to gather and is not relevant for this work.

Code Coverage is used to measure the degree to which code has been tested [15].

Of all the software metrics analysed, those proposed by *Chidamber and Kemerer* [42] are the most well-known and generally accepted metrics amongst software engineers. They propose a set of six metrics for object oriented languages. These metrics were developed in an attempt to give software engineers an impartial insight to help them judge the quality of their own system. In research, they have been used in important projects such as the DaCapo project [20] to analyse different aspects of the benchmarks and compare them with other benchmarks such as jvm98. These metrics are described in Table 11.

Spinellis [145] proposed the addition of two other software metrics, namely *afferent coupling*, and *number of public methods*. These metrics are described in

weighted methods per class (WMC): This metric is simply a count of the number of methods defined by the current class.

depth of inheritance tree (DIT): The `java.lang.Object` class has a DIT score of 1. The DIT score increases by 1 for every edge on the path through the inheritance tree from the root to the current class.

number of children (NOC): The number of classes that are immediate subclasses of the current class.

coupling between object classes (CBO): The number of classes upon which the current class depends. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

response for a class (RFC): Ideally, this should measure the number of different methods that can be executed when an instance method of the current class is invoked, summed over all instance methods for this class. This would involve calculating the transitive closure of the method's call graph, which has the potential to be expensive and inaccurate. Instead, the metrics measurement tool calculates a rough approximation to RFC by simply counting method calls within the class's method bodies. This simplification was also used in the original CK metrics paper [42].

lack of cohesion in methods (LCOM): This counts sets of methods in the current class that are not related through the sharing of some of the class's fields. The metrics measurement tool considers all pairs of the class's methods. In some of these pairs, both methods access at least one common field of the class, while in other pairs the two methods do not access any common fields. The LCOM score is calculated as the difference between the number of pairs that do share at least one field from the number of pairs that do not share any fields.

Table 11: The Chidamber and Kemerer metrics suite.

afferent couplings (AC): This measures the number of classes that depend on the current class. Coupling is defined in the same way as for CBO above. Conceptually, Ca is the inverse of CBO.

number of public methods (NPM): This is simply a count of the methods in the current class that are declared as public. It can be used to give an indication of the size of an API provided by a package.

Table 12: Spinellis's additional metrics

table 12.

This chapter, combines the use of Chidamber and Kemerer's metrics with Spinellis's metrics. This set of 8 metrics are referred to as the *CK* metrics suite.

5.1.3 CK Metrics

CK software metrics [42] target object oriented languages and identify relationships between different classes of a program. These metrics were developed to address the criticism expressed towards previous metrics regarding drawbacks such as their lack of theoretical basis and the fact that they are too difficult to collect. The authors provide a tool capable of automatically calculating each metric's value by analysing the bytecode of the target Java program. They may be used easily by managers to track software development progress.

More generally, they describe certain aspects of interactions between classes. This is important for this work because studying the way classes interact together could provide indications of how long objects of certain types generally live. For instance, one could reasonably assume that objects of two classes with high coupling (CBO) are likely to die at the same time.

CK software metrics have recently been used in the GC field. On the basis that no garbage collector is the best for every program, Singer used CK metrics to predict what garbage collector is most appropriate for a particular program [141].

5.2 Information Theoretic Analysis

This section provides a theoretic analysis of the correlation between software metrics and object lifetimes. Please note that this section is based on the paper "An Information Theoretic Evaluation of Software Metrics for Object Lifetime Prediction" [142] which was largely written by Jeremy Singer from the University of Manchester with whom the author was collaborating on this research (see Appendix A). The content of this section originally written by Singer is provided because of its relevance to this thesis.

5.2.1 NMI Values

In order to find which metrics could be good lifetime indicators, the correlation between the lifetime of an object and each of the eight possible source metrics and the eight possible destination metrics is calculated. This is done using Normalized Mutual Information (NMI), also known as *transinformation*, an information theory measure calculating the dependence of two variables. A high value (close to 1) indicates a good correlation, whereas a value close to 0 indicates a bad correlation.

Table 13 shows the results obtained for the NMI-based correlation of single features with lifetime.

Table 13 shows relatively bad correlation between metrics and lifetime. The metric which correlates the most with object lifetimes is the source LCOM, which measures the coheasion (or lack of) between methods. Intuitively, this makes sense as objects with a high cohesion are likely to have related lifetimes. However, with a highest NMI of 0.37, using only one metric to predict lifetime is unlikely to produce good results.

This table also shows an interesting result: the two metrics with the highest correlation are two source metrics, which suggest that considering the source of an object is an important criteria. Most type-based lifetime studies have consisted on analysing the destination object and hence, this is an important discovery. As explained in Chapter 3, the only published research we are aware of which

metric	*NMI*
lifetime	1.000
source LCOM	0.370
source RFC	0.342
dest LCOM	0.324
dest RFC	0.314
dest NPM	0.261
dest WMC	0.257
source WMC	0.233
source CBO	0.180
source NPM	0.163
source Ca	0.117
dest DIT	0.079
dest Ca	0.056
source DIT	0.037
source NOC	0.035
dest CBO	0.030
dest NOC	0.012

Table 13: NMI-based correlation of single features with lifetime

considers the use of source object as well as the destination object is the paper of this author [95].

For more information regarding the way these measures were obtained, please refer to Appendix A.

5.2.2 Combining Metrics

The previous section showed that metrics individually do not correlate well with lifetimes. However, it would be interesting to see if using combinations of metrics may be more suited to predicting object lifetimes. This section analyses the correlation between combinations of metrics.

Table 14 shows the results of the cross-correlation analysis. In this table each column and each row correspond to a particular metric (apart from lifetime). This allows us to see the correlation of metrics with respect to each other considering the lifetime. As expected, comparing a metric with itself gives an NMI value of

	sWMC	sDIT	sNOC	sCBO	sRFC	sLCOM	sCa	sNPM	dWMC	dDIT	dNOC	dCBO	dRFC	dLCOM	dCa	dNPM	lifetime
sWMC	1.000	0.354	0.475	0.513	0.716	0.752	0.498	0.616	0.282	0.232	0.184	0.232	0.316	0.343	0.291	0.310	0.233
sDIT		1.000	0.045	0.271	0.474	0.515	0.161	0.288	0.133	0.054	0.042	0.040	0.186	0.170	0.097	0.134	0.037
sNOC			1.000	0.259	0.588	0.645	0.217	0.373	0.174	0.068	0.020	0.062	0.276	0.315	0.082	0.312	0.035
sCBO				1.000	0.633	0.612	0.462	0.396	0.176	0.116	0.072	0.226	0.278	0.242	0.312	0.185	0.180
sRFC					1.000	0.701	0.583	0.661	0.371	0.293	0.272	0.298	0.416	0.431	0.365	0.391	0.342
sLCOM						1.000	0.607	0.686	0.381	0.299	0.269	0.309	0.421	0.446	0.352	0.403	0.370
sCa							1.000	0.393	0.175	0.069	0.061	0.186	0.276	0.233	0.266	0.177	0.117
sNPM								1.000	0.236	0.207	0.151	0.164	0.291	0.297	0.216	0.267	0.163
dWMC									1.000	0.335	0.458	0.411	0.719	0.669	0.425	0.691	0.257
dDIT										1.000	0.321	0.067	0.419	0.506	0.075	0.374	0.079
dNOC											1.000	0.133	0.616	0.645	0.147	0.548	0.012
dCBO												1.000	0.497	0.455	0.367	0.339	0.030
dRFC													1.000	0.687	0.488	0.676	0.314
dLCOM														1.000	0.432	0.661	0.324
dCa															1.000	0.387	0.056
dNPM																1.000	0.261
lifetime																	1.000

Table 14: Cross-correlation of features with other features.

1, meaning the correlation is perfect. The closer the NMI is to 1, the higher the correlation between the two metrics.

This analysis shows that source metrics correlate highly with other source metrics, while destination metrics correlate highly with other destination metrics. On the other hand, there is a low correlation between source metrics and destination metrics. Therefore, a combination of source metrics and destination looks most appropriate to predict object lifetimes.

5.2.3 Conditional Mutual Information Maximisation

Table 13 revealed that single features all have low NMI scores. Therefore, a *combination* of features is required to obtain reliable predictions.

However selecting features based only on NMI does not take any cross-correlation between features into account. Ideally, metrics that have a high individual correlation with lifetime, and have low cross-correlation with each other should be selected.

Fleuret in [56] presents an attractive algorithm known as *Conditional Mutual Information Maximisation* (CMIM) that automatically selects the most appropriate features (metrics in this case). The approach iteratively chooses features that maximise their mutual information with the class to predict (lifetime in the context of this work), conditioned on features already picked. This CMIM criterion does not select a feature *similar* to already picked ones, even if it is individually informative, since such a similar feature does not carry *additional* information about the class to predict.

Conditional mutual information is calculated as:

$$I(U;V|W) = H(U|W) - H(U|W,V) \tag{5}$$

This value is an estimate of the quantity of information shared between U and V when W is known. If V and W carry the same information about U, then the two terms on the right are equal and the conditional mutual information is

zero, even if both V and W are individually informative. Conversely if V contains information about U which is not present in W, then the difference is large and the conditional mutual information is high.

The CMIM algorithm operates as follows. It aims to pick k features from a total of n, in order of relevance with the most relevant feature first. Incidentally, if CMIM is used to select n features from n, then a relevance-ordered ranking of the entire feature set which takes into account cross-correlations happens in a way that the simple ranking based on NMI scores in Table 13 did not.

The algorithm maintains a score vector, with one element for each feature. Initially, the score vector is set up so $score[i]$ contains the *unnormalized* mutual information score for the ith feature (with the lifetime). At each iteration, the feature with the highest score value is taken as the next selected feature. Then the score vector is recomputed, with each element $score[i]$ set to the minimum value of $(score[i], I(\text{lifetime}; i\text{th feature}|\text{last selected feature}))$. This ensures that $score[i]$ is low if at least one of the features already picked is similar to the ith feature.

His algorithm is implemented and uses to rank each metric in order of relevance for object lifetimes prediction. Table 15 gives the results of this CMIM analysis. Note that in the table, each score is the highest value in the score vector at that particular iteration, for a feature that has not been selected by previous iterations.

This table confirms that considering the source when predicting the lifetime of an object is important. As with the NMI values, the most relevant metric is once again source LCOM. If it was decided to use only a limited amount of metrics to predict lifetime, metrics with the highest values should be used.

Experimental results are presented in the following sections of this chapter.

5.3 Using Scalar Metrics

In the first set of experiments, a combination of 8 source metrics and 8 destination metrics were used. Metrics values were used as they were reported by Spinellis's

metric	*CMIM score*
source LCOM	0.407
dest RFC	0.345
source RFC	0.177
source CBO	0.144
dest LCOM	0.142
source NPM	0.113
source WMC	0.104
source Ca	0.086
dest NPM	0.085
dest WMC	0.072
dest Ca	0.061
source DIT	0.041
dest DIT	0.038
dest CBO	0.033
source NOC	0.025
dest NOC	0.007

Table 15: CMIM-based ranking of features for prediction of object lifetime.

metrics tool. These values were then fed into Clementine, which generated rules that were incorporated into a metrics-lifetime knowledge bank (see Chapter 3).

Results of these experiments are reported below.

5.3.1 Using All Metrics

This section reports the results obtained using all metrics as predictors. In contrast, Section 5.3.2 uses only a subset of the metrics. Figures 10, 11, 12 and 13 show graphs reporting the GC time performance improvement (in milli-seconds) of the system when compared with a default run of each benchmark, with a default JVM which does not implement the advice system (see Chapter 3).

In these graphs, +n% would mean that the benchmark is running n% slower with the advice system than it would otherwise. In other words, positive numbers mean the system is degrading GC time performance while negative numbers indicate improvements.

These four graphs show different degrees of conservatism with confidence levels

varying from 75% to 99%.

These graphs show significant performance degradation on most benchmarks and at most heap sizes. Varying the confidence level to a higher threshold slightly improves performance, but the advice system remains harmful.

Next, the impact on performance is studied by using only the four metrics having the best conditional mutual information (CMIM score see Section 5.2.3) with the data. They are: source LCOM, destination RFC, source RFC, and source CBO (see Table 23).

5.3.2 Top 4 metrics

In an attempt to improve the quality of the advice, this section considers only the top 4 metrics with regards to their CMIM values. Figures 14, 15, 16 and 17 show graphs reporting the GC time performance improvement of the system, using only the top four scalar metrics, when compared with a default JVM which does not implement the advice system (see Chapter 3).

Once again, these graphs show performance degradation at most confidence levels (75%, 85% and 95%). At confidence 99% however, the system shows improvements of up to 9% for '_228_jack' while '_213_javac' behaves badly and shows performance degradations of up to 43%. The advice is mostly neutral for other benchmarks.

5.3.3 Analysis

These graphs show us that obtaining GC time speedups using these software metrics is rarely effective. In most instances, the advice generated from metrics data is harmful to the system, with the exception of graph 17 where results are reported using only the top 4 metrics at 99% confidence. In this case, performance improvement is seen for just one benchmark, '_228_jack', of up to 9%.

During this research, the segregation of sites allocating scalar objects from sites allocating arrays was explored, but such segregation did not help producing

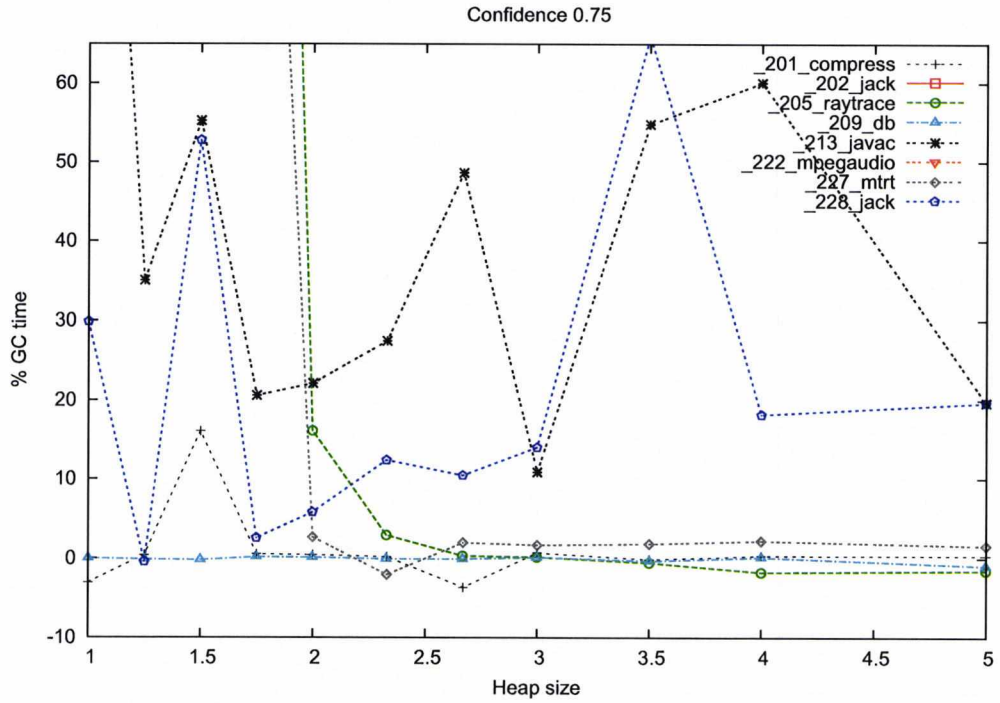


Figure 10: GC time improvement, all metrics, all sites, Speed 100, confidence 75%.

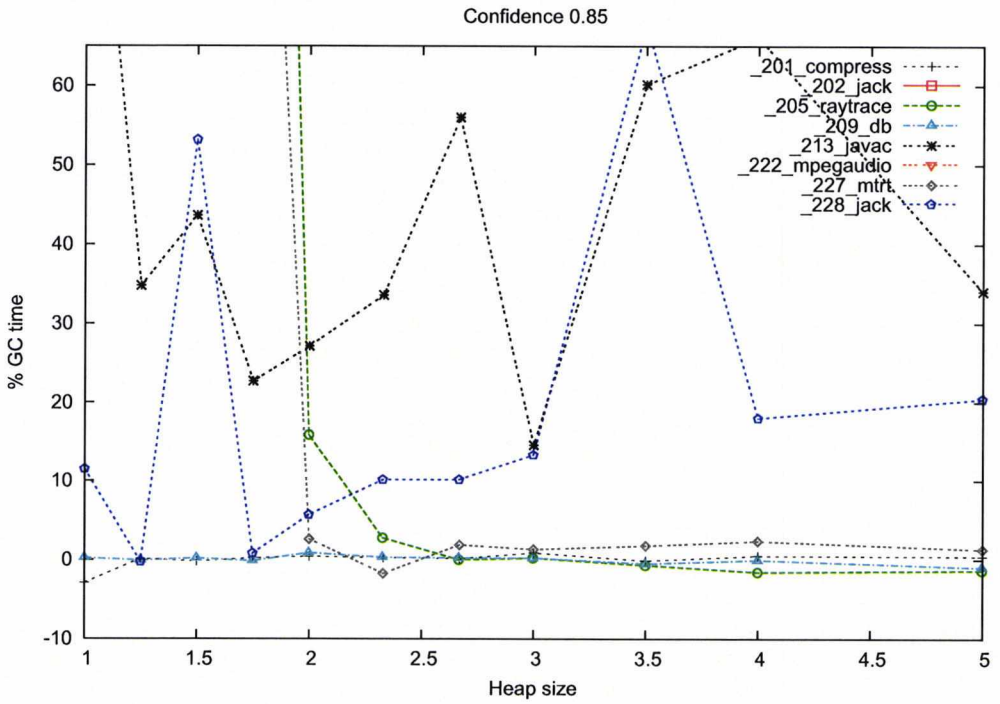


Figure 11: GC time improvement, all metrics, all sites, Speed 100, confidence 85%.

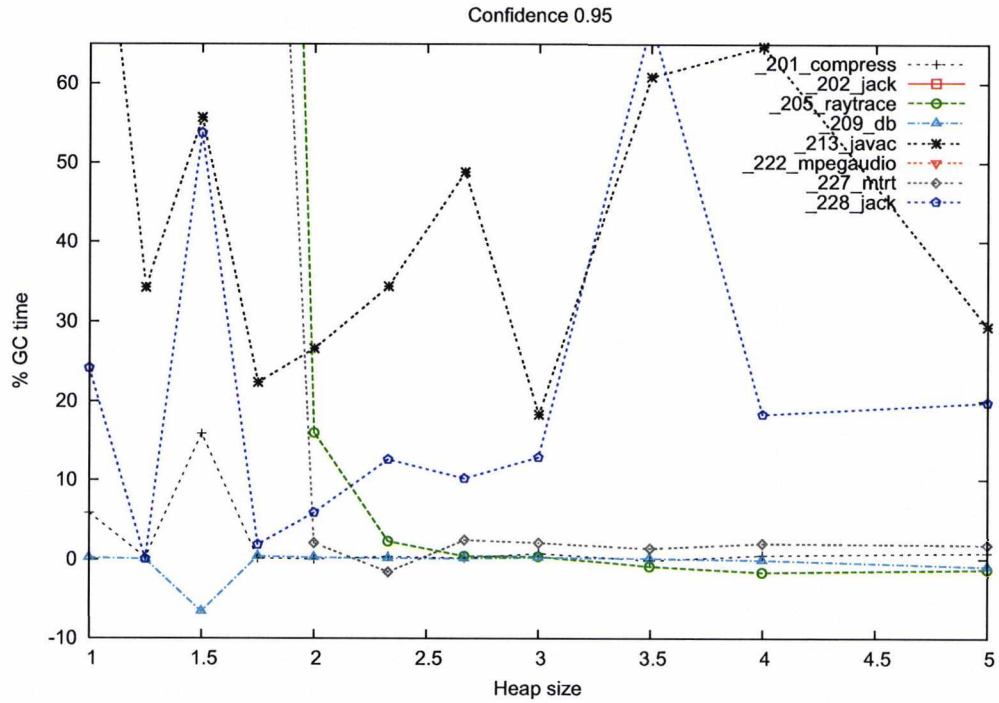


Figure 12: GC time improvement, all metrics, all sites, Speed 100, confidence 95%.

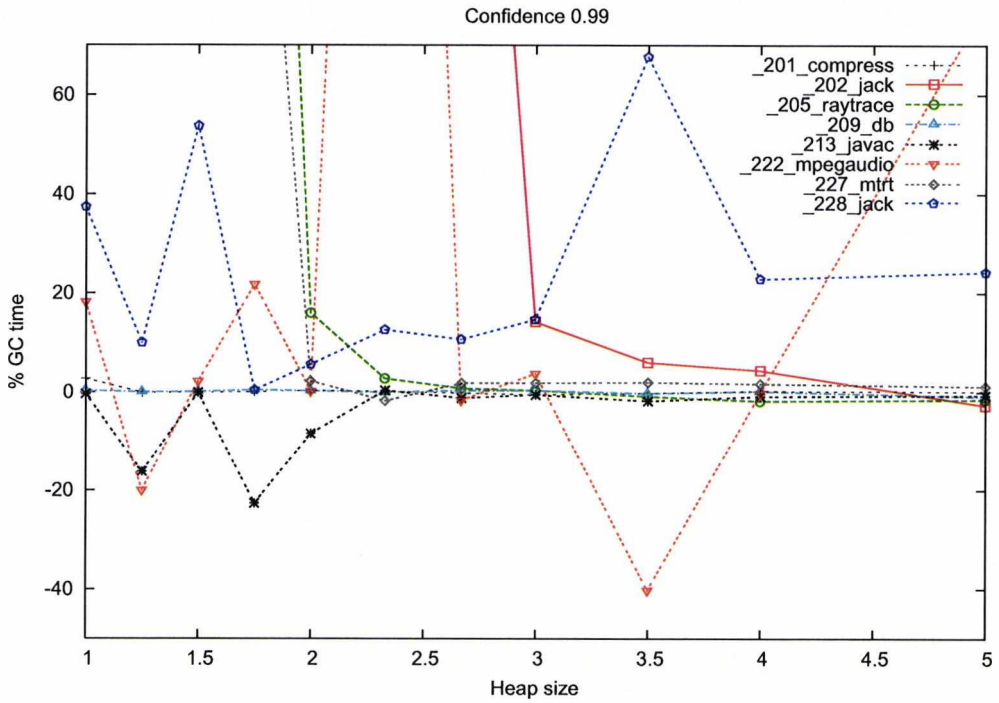


Figure 13: GC time improvement, all metrics, all sites, Speed 100, confidence 99%.

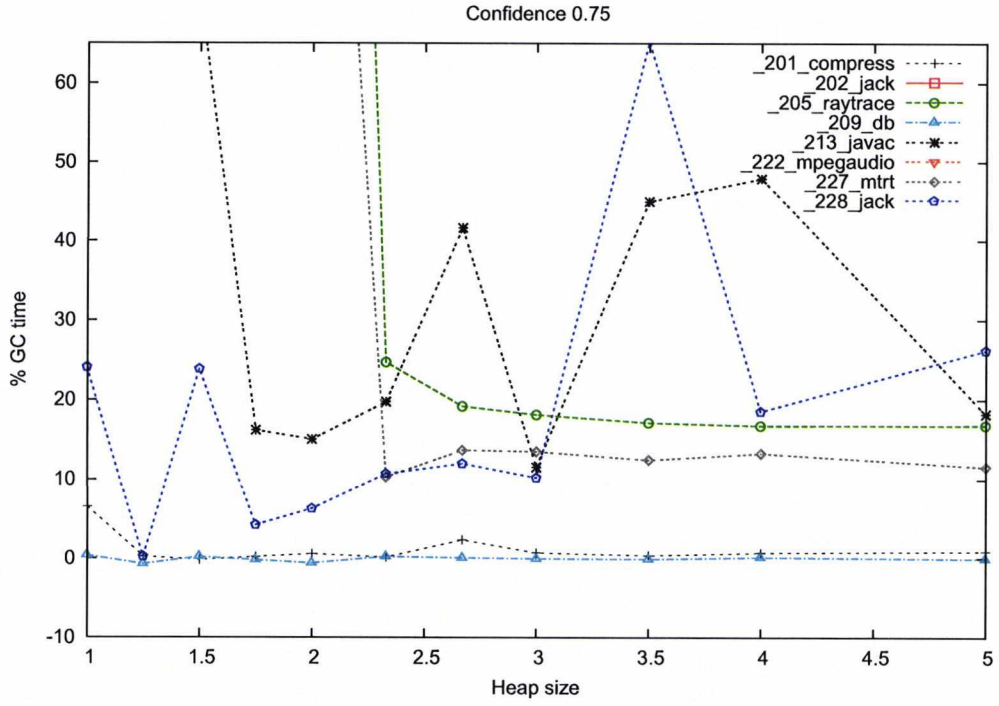


Figure 14: GC time improvement, top 4 metrics, all sites, Speed 100, confidence 75%.

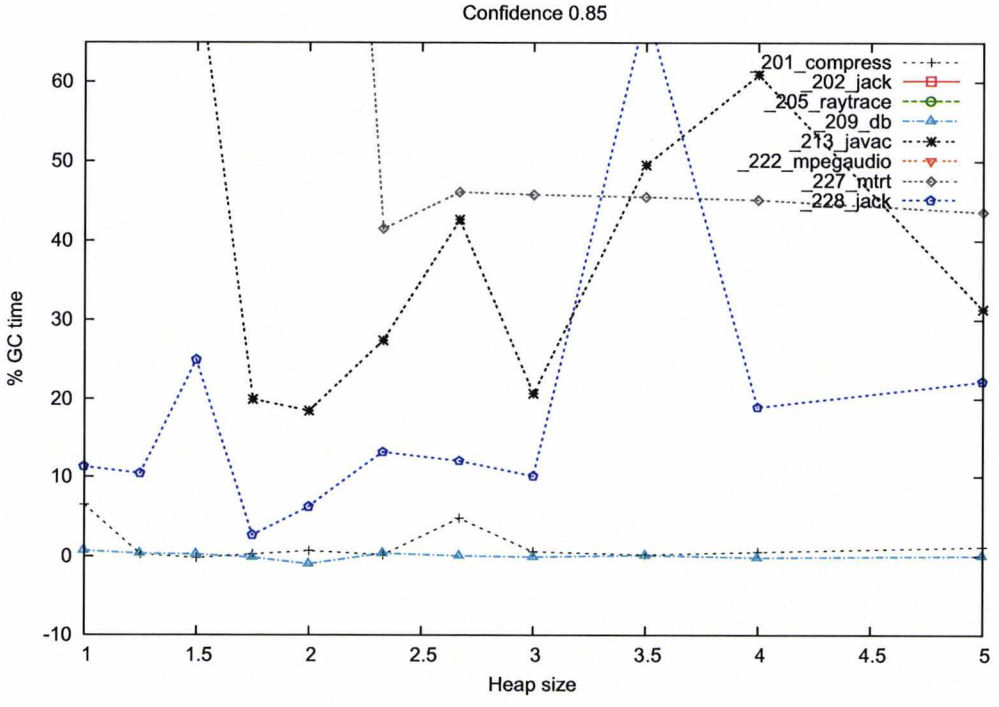


Figure 15: GC time improvement, top 4 metrics, all sites, Speed 100, confidence 85%.

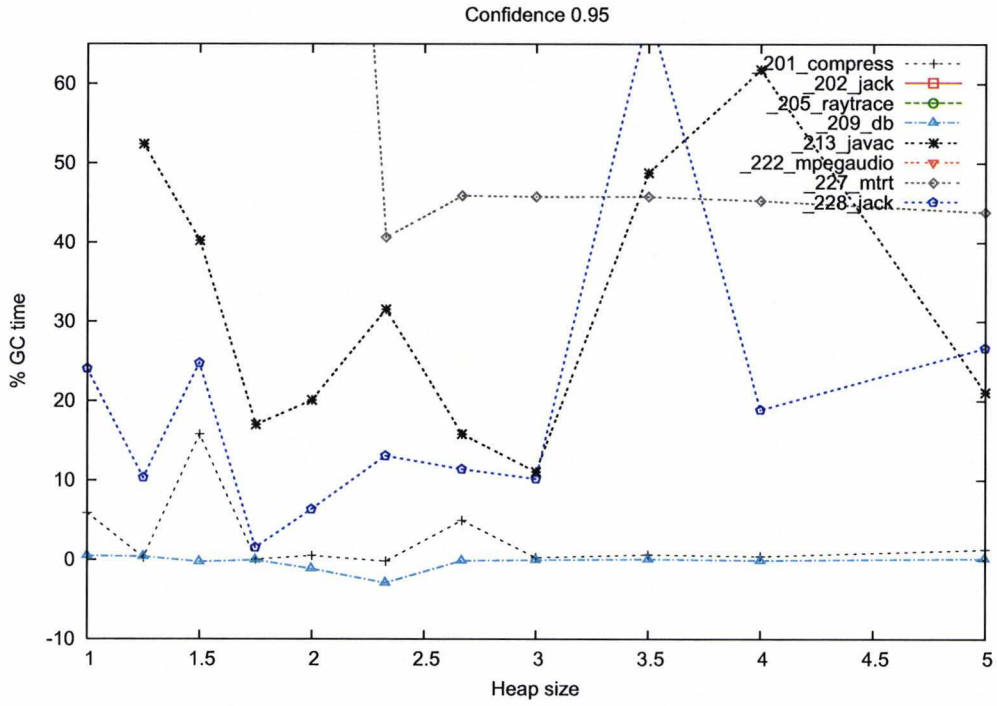


Figure 16: GC time improvement, top 4 metrics, all sites, Speed 100, confidence 95%.

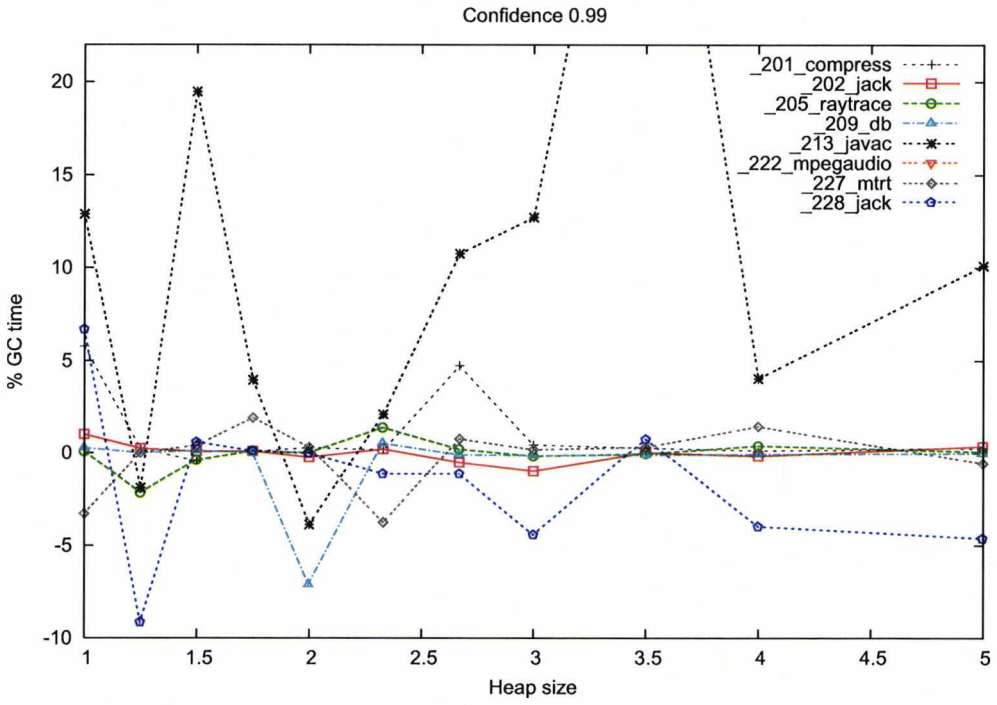


Figure 17: GC time improvement, top 4 metrics, all sites, Speed 100, confidence 99%.

advice leading to better performance. These results are not reported in this thesis.

5.4 Using Ternary Values

One of the problems of generating advices using the raw metrics values as done the previous section, is the disparity of values. Ranges of values for each metric are generally very large and generating advice based on these values is imprecise and might lead to misclassifications. For example, values for the WMC metric can range from 0 to 1300.

The way metrics ranges could be turned into ternary data in order to ease classification is explored in this section. First, the mean and the standard deviation (SD) of each metric is calculated for the training set (the DaCapo benchmarks). Then, each object's metrics is classified in 3 categories:

- *Low*: allocation sites allocating objects whose metric value is more than $n * SD$ below the calculated mean.
- *High*: allocation sites allocating objects whose metric value is more than $n * SD$ above the calculated mean.
- *Middle*: allocation sites allocating objects whose metric value is between the *Low* and the *High* boundaries.

The results obtained using advice generated using metrics ternary values, for different values of n are analysed below.

Figures 18, 19, 20 and 21 show the results obtained using all 16 metrics (8 source metrics and 8 destination metrics) at confidence levels of respectively 75%, 85%, 95% and 99% using $n = 0.25$.

Using the top 4 metrics only, the data-mining algorithm was unable to find any rule with a confidence higher or equal to 85%. At confidence 75%, the system was able to generate some advice for only 3 benchmarks, but the advice degrades performance. For this reason, no figures are reported for the top 4 metrics.

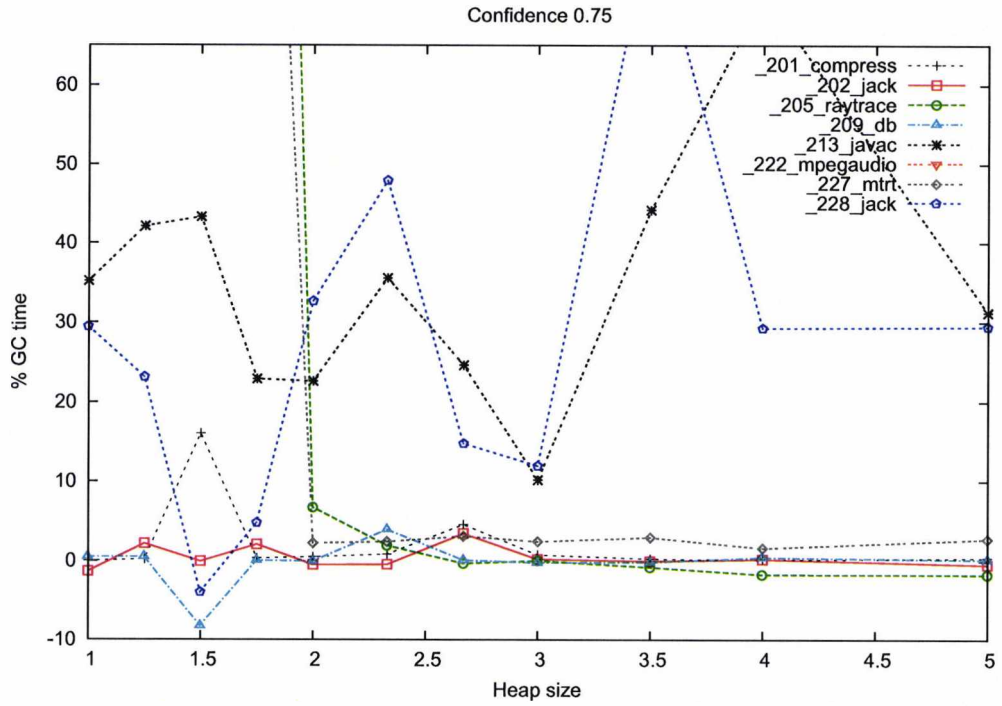


Figure 18: GC time improvement, all metrics, all sites, ternary classification N=0.25, Speed 100, confidence 75%.

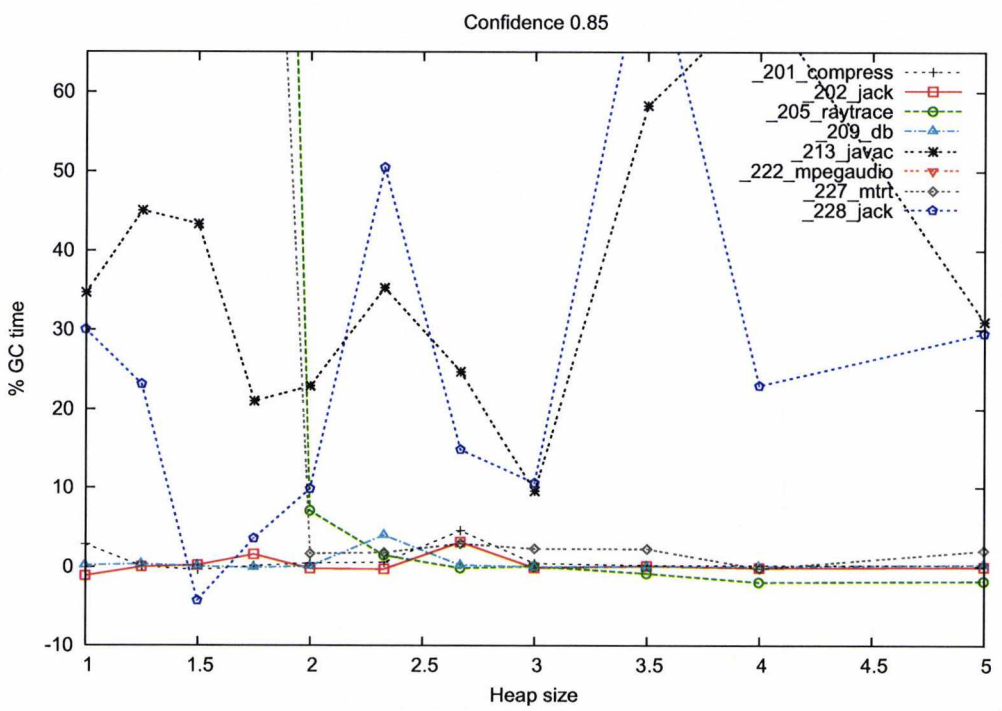


Figure 19: GC time improvement, all metrics, all sites, ternary classification N=0.25, Speed 100, confidence 85%.

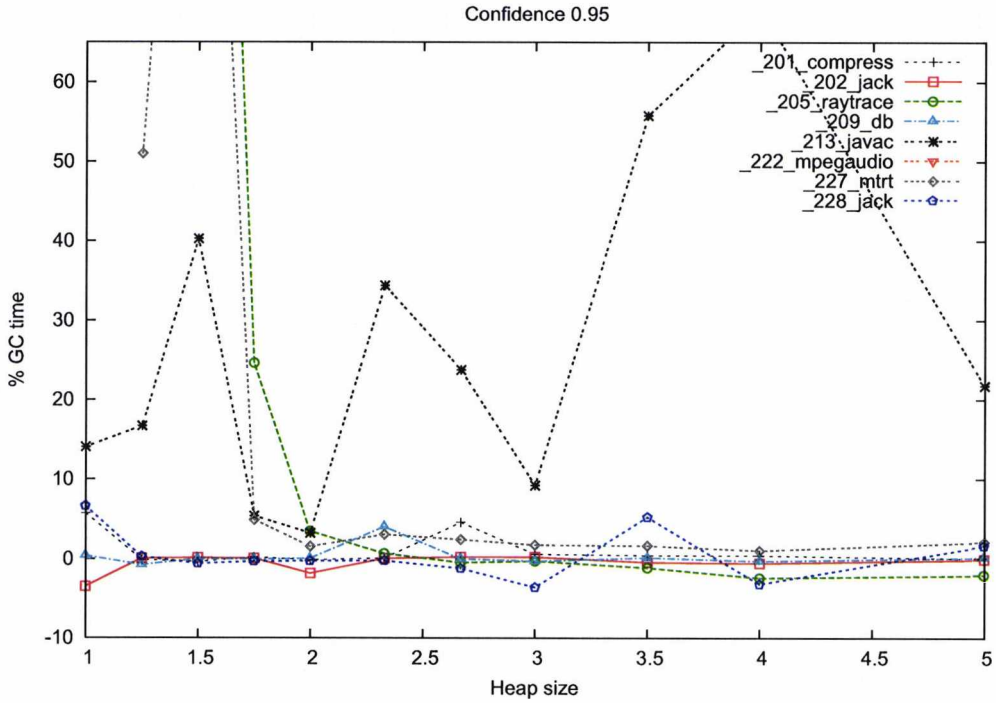


Figure 20: GC time improvement, all metrics, all sites, ternary classification N=0.25, Speed 100, confidence 95%.

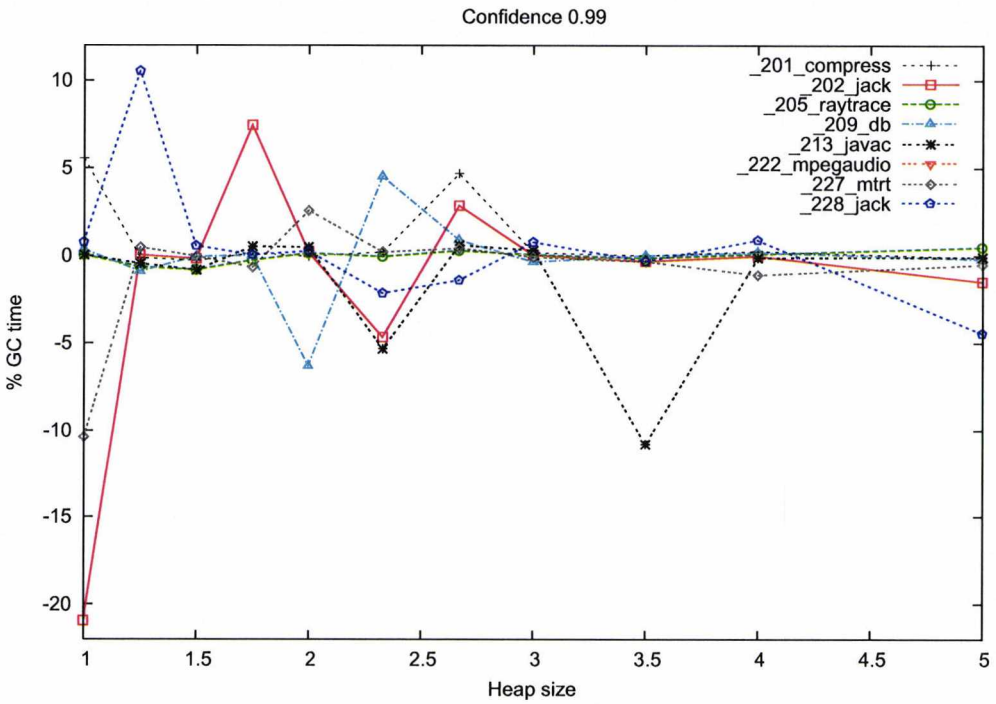


Figure 21: GC time improvement, all metrics, all sites, ternary classification N=0.25, Speed 100, confidence 99%.

Figures 22, 23 and 24 show the results obtained using all 16 metrics (8 source metrics and 8 destination metrics) at confidence levels of respectively 75%, 85% and 95%, using $n = 1$. The data-mining algorithm was unable to find any rule with a confidence equal or higher than 99%.

These graphs show that GC time performance is greatly affected by the advice. For most tests, results are not satisfactory as GC time degrades on some benchmarks, and improves few.

However, of all these tests, one in particular, seem to stand out with slightly better results than others. Using ternary values with *n=100* at 95% confidence shows GC time improvements in '_213_javac' and '_222_mpegaudio'.

In general ternary values produced more stable results (less jumps in performance) than using raw values, but performance results are still disappointing.

5.5 Using Inter-Quantile Values

Succi et al. advocate using median and inter-quartile ranges for discretizing CK metrics data [151]. They demonstrate how the use of mean and standard deviations can create problems such as unrealistic values and impossible fractional limits.

In this section inter-quartile values are used to segregate each metric into one of three categories: high (+1), middle (0) and low (-1). For each metric at each allocation site, the classification scheme works as follows:

- A value of -1 is given if the inter-quartile value of a metric is in the bottom 25% of the range of values for that metric.
- A value of 0 is given if the inter-quartile value of a metric is in middle 50% (i.e. close to median).
- A value of +1 is given if the inter-quartile value of a metric is in the top 25% of the range of values for that metric.

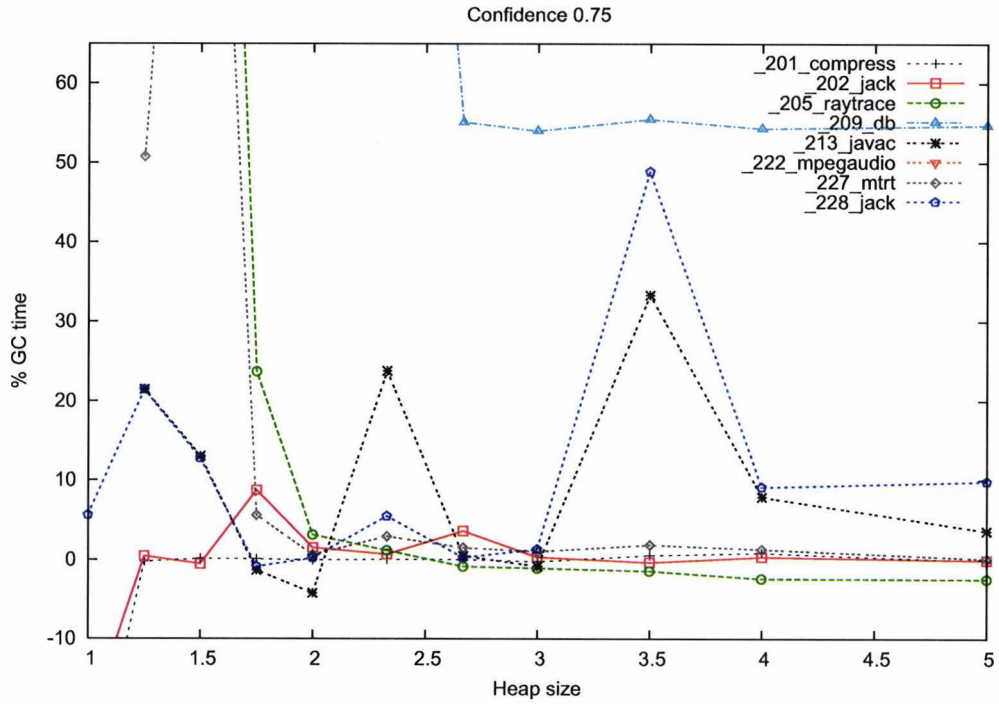


Figure 22: GC time improvement, all metrics, all sites, ternary classification N=1, Speed 100, confidence 75%.

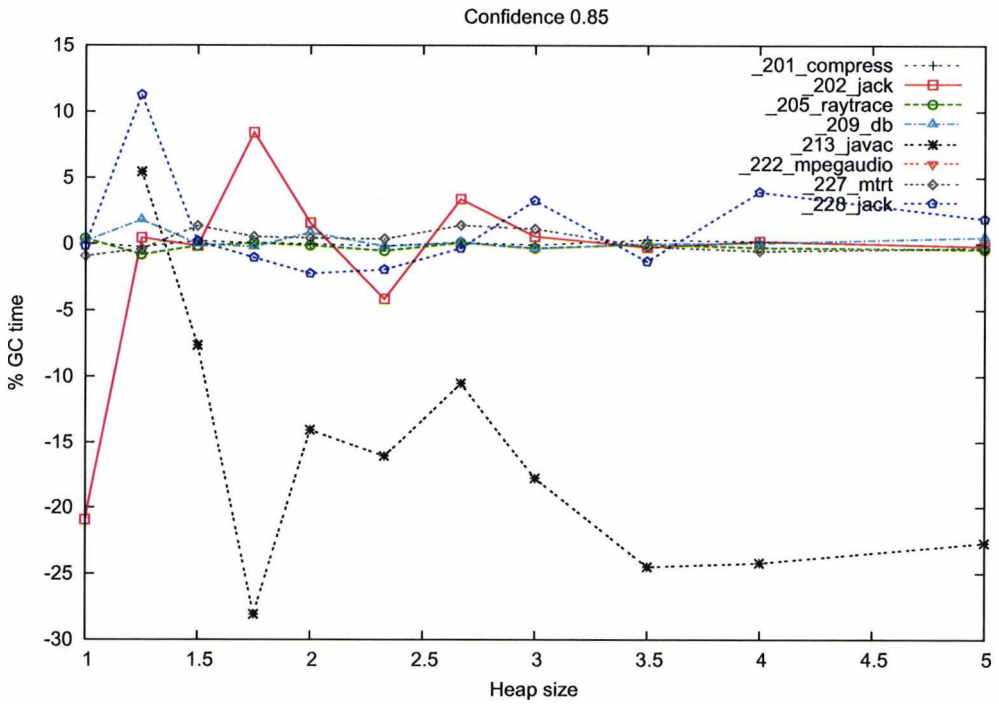


Figure 23: GC time improvement, all metrics, all sites, ternary classification N=1, Speed 100, confidence 85%.

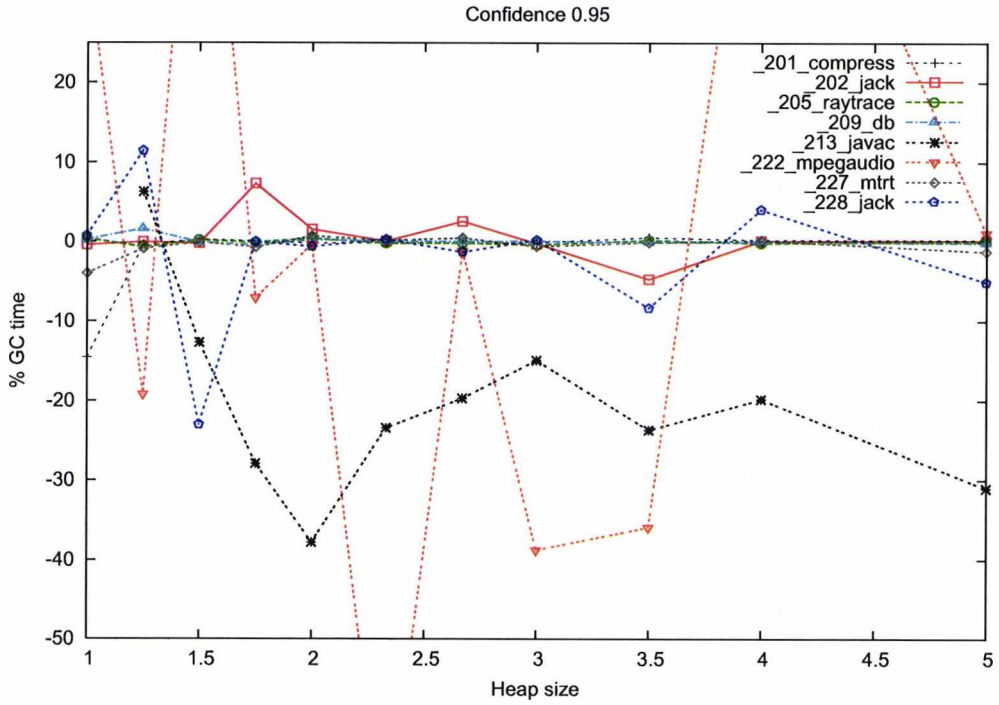


Figure 24: GC time improvement, all metrics, all sites, ternary classification N=1, Speed 100, confidence 95%.

Figure 25 reports the results using a confidence of 75%. In this graph, '_222_mpeg-audio' does not appear because the VM ran out of memory due to the advice pretenuring too much data. '_202_jess' however did run completely, but does not appear either because the time overhead was too large to fit on the graph. In general, this test shows no improvements and makes the GC slower.

Figure 26 reports the results using a confidence of 85%. Again, '_222_mpeg-audio' made the VM run out of memory due to the advice pretenuring too much data. This shows slightly better performance than Figure 25 but still degrades performance.

Figure 27 reports the results using a confidence of 95%. Here, all benchmarks ran properly, performance is reported for all of them. '_205_raytrace' and '_227_mtrt', as well as '_213_javac' at small heap sizes show important degradations. The advice is mostly neutral to other benchmarks.

Finally, Figure 28 reports results using a confidence of 99%. Here, results are better than at lower confidences, but results are not stable and big jumps in performance can be observed. Again, results show mostly degradation.

5.6 Conclusion

In this chapter, the use of Chidamber and Kemerer's software metrics was experimented to derive lifetime predictions. These metrics are attractive because they can be gathered easily by analysing program source code using Spinellis's tool. Furthermore, this tool provides us with real numbers for each metric (for both source and destination) that can be associated with each allocation site.

Unfortunately, the information theoretic analyses showed little correlation between metrics and lifetime. The experimental results confirmed these findings by performing poorly. This Chapter attempted to discretise efficiently the data-set into meaningful values using metrics means and inter-quartile values. This Chapter also attempted using only a subset of all the metrics, but both these attempts were unsuccessful. It is therefore concluded that CK software metrics are not

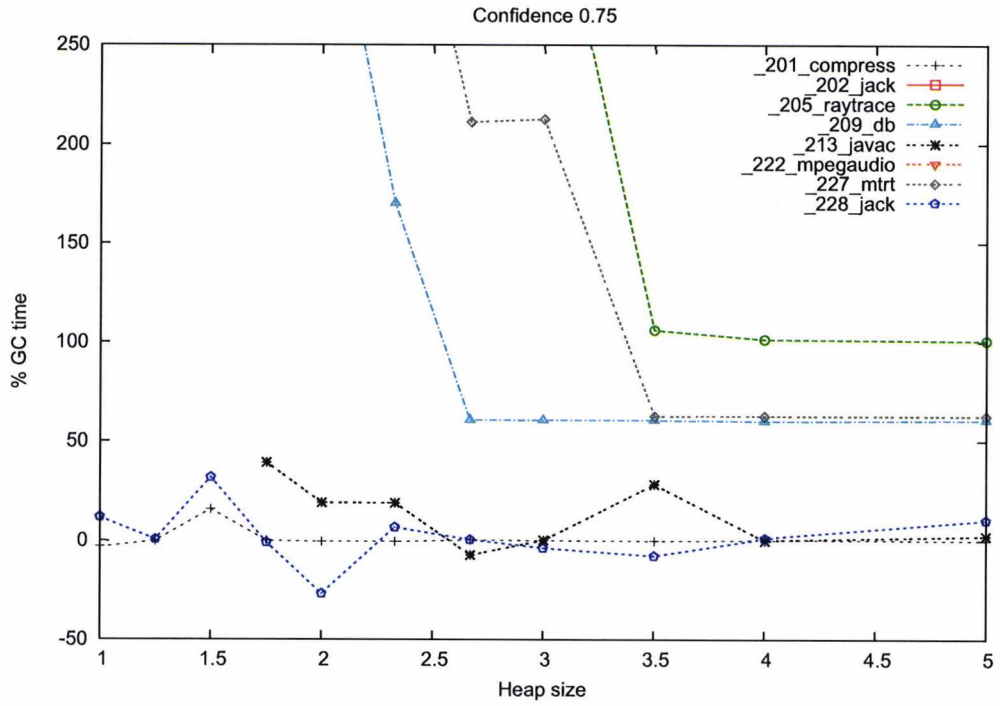


Figure 25: GC time improvement, all metrics, all sites, inter-quartile, Speed 100, confidence 75%.

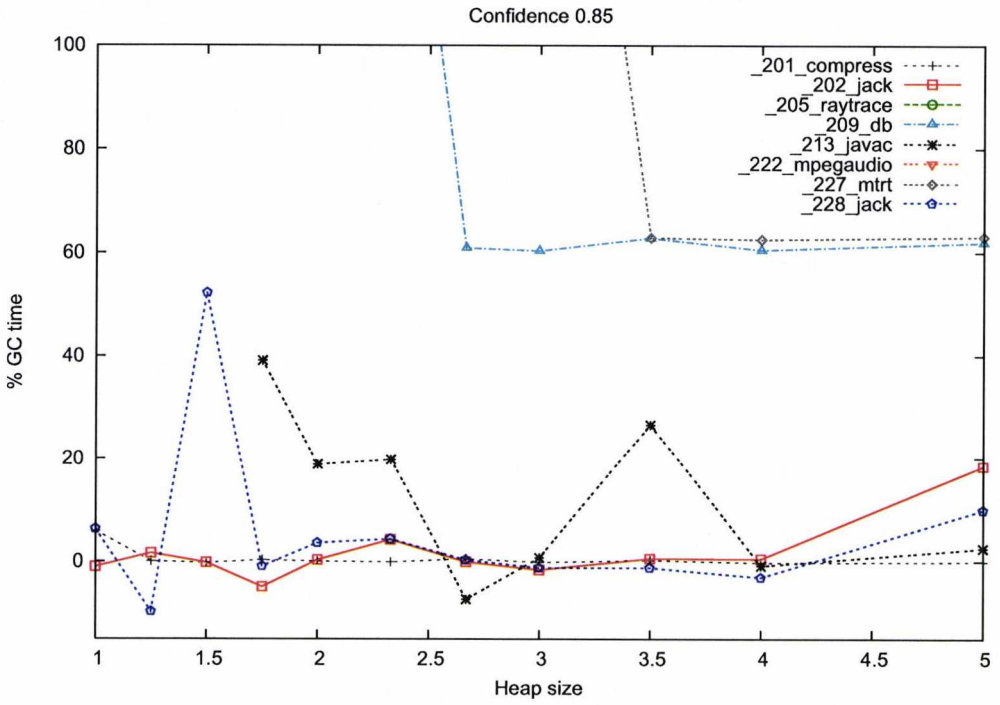


Figure 26: GC time improvement, all metrics, all sites, inter-quartile, Speed 100, confidence 85%.

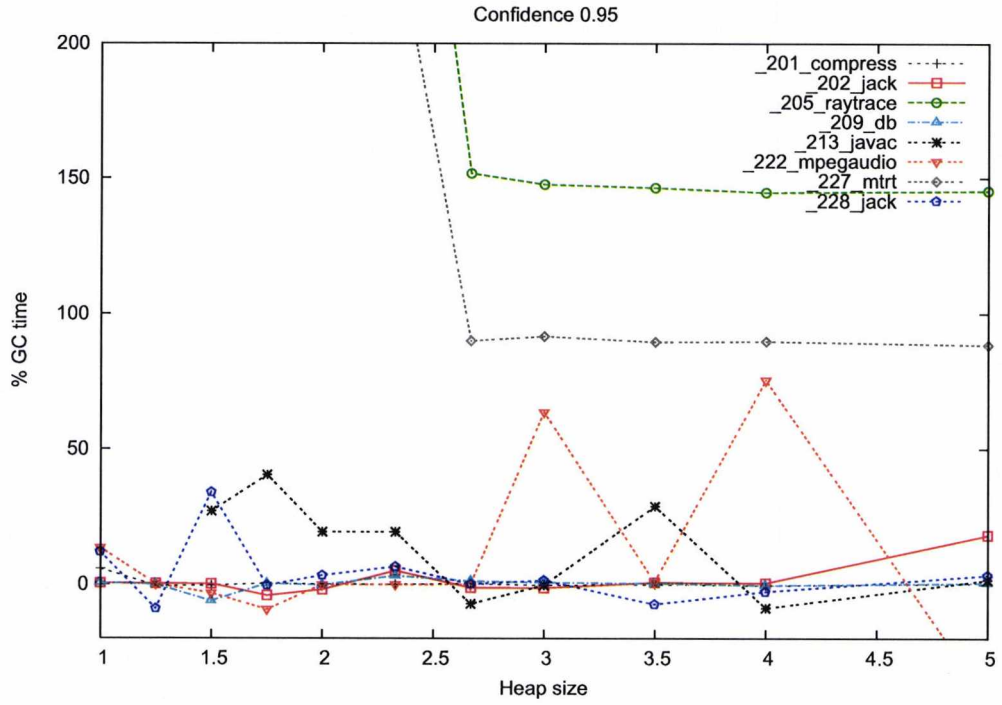


Figure 27: GC time improvement, all metrics, all sites, inter-quartile, Speed 100, confidence 95%.

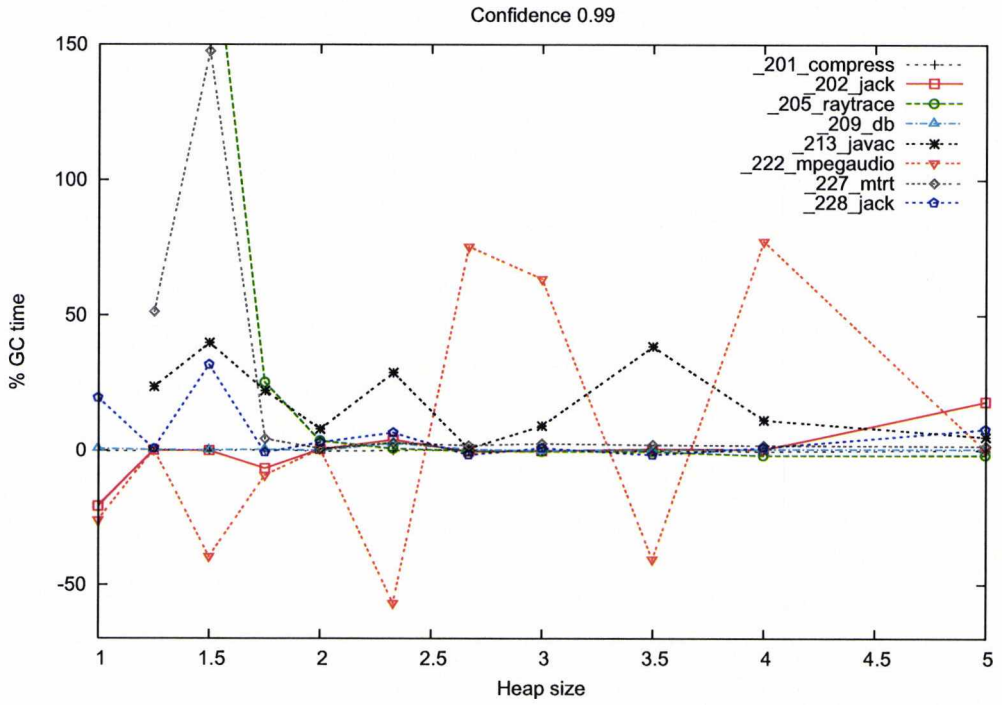


Figure 28: GC time improvement, all metrics, all sites, inter-quartile, Speed 100, confidence 99%.

good lifetime predictors.

The fact that CK metrics are not good predictors can be explained by a combination of several factors specific to software metrics.

Firstly, software metrics ranges can be very wide. For example, the RFC metric in the training set ranges from 1 to 1,370, where class "*com/ibm/JikesRVM/VM_Assembler.java*" as an RFC value of 1,370. Likewise, the WMC metric ranges from 1 to 1,303, where that same class has a value of 1,303. The LCOM metric ranges from 1 to 16214, where class "*com/ibm/JikesRVM/VM_BaselineCompiler*" has a value of 16,214.

This makes classification difficult, as the algorithm has to decide which ranges of values correlate best to specific lfietimes. This is unfortunately imprecise and can lead to misclassifications.

Secondly, CK Metrics are general purpose metrics. They were created in order to produce a concise and quantitative analysis of certain aspects of a software system. They define and quantify interactions between classes. However, they offer no insight on the programmer's intent as they do not focus on class level properties, but rather on inter-class relationships.

It is possible that lower-level metrics could give a better indication of object lifetimes. The following chapter analyses the use of micro-patterns to derive predictions.

Chapter 6

Using Micro-Patterns

The previous chapter explored the use of software metrics to predict object lifetimes. In this chapter, a new approach taking advantage of class-level information and developers coding practices is analysed via the use of *Micro-Patterns* [62].

The results are summarised in our paper *Decrypting The Java Gene Pool* published at the International Symposium on Memory Management in 2007 [95].

The first section of this chapter describes micro-patterns, what they are and what this work uses them for. The following sections show the experimental results and the effect on GC time (both overall and pause time distribution) of the pretenuring scheme. The effect of pretenuring on overall run-time, and on pause time is then studied. Finally, object placement (number of sites identified for pretenuring, volume pretenured, etc.) and the accuracy of the predictions are analysed.

6.1 Background

Every human in the planet is unique. Attributes like hair colour, eyes colour, skin colour etc. are encoded into human genes. Likewise, in Java, each class is unique and has its own characterisctis. In 2005, Gil and Maman [62] introduced *micro-patterns*, which allow the mechanical characterisation and classification of class features.

	Micro-pattern	*Definition*	All *Src-Dst*	SPEC *Src-Dst*
Degenerate classes	*Designator*	Interface with no members.	0-0	0-0
	Taxonomy	Empty interface extending another interface.	4-0	1-0
	Joiner	Empty interface joining two or more superinterfaces.	0-1	0-0
	Pool	Class which declares only static final fields, but no methods.	8-0	3-0
	Function Pointer	Class with a single public instance method, but with no fields.	1-3	1-1
	Function Object	Class with a single public instance method, and at least one instance field.	9-5	4-3
	Cobol Like	Class with a single static method, but no instance members.	1-0	0-0
	Stateless	Class with no fields, other than static final ones.	15-6	5-4
	Common State	Class in which all fields are static.	18-0	7-0
	Immutable	Class with several instance fields, which are assigned exactly once, during instance construction.	5-6	3-3
	Restricted Creation	Class with no public constructors, and at least one static field of the same type as the class.	6-4	5-2
	Sampler	Class with one or more public constructors, and at least one static field of the same type as the class.	8-9	7-3
Containment	Box	Class which has exactly one, mutable, instance field.	6-14	2-10
	Compound Box	Class with exactly one non primitive instance field.	32-22	17-12
	Canopy	Class with exactly one instance field that it assigned exactly once, during instance construction.	4-13	2-6
	Record	Class in which all fields are public, no declared methods.	0-0	0-0
	Data Manager	Class where all methods are either setters or getters.	1-10	0-5
	Sink	Class whose methods do not propagate calls to any other class.	11-9	4-5
Inheritance	Outline	Class where at least two methods invoke an abstract method on `this`.	7-4	5-2
	Trait	Abstract class which has no state.	4-0	2-0
	State Machine	interface whose methods accept no parameters.	0-0	0-0
	Pure Type	Class with only abstract methods, and no static members, and no fields.	0-0	0-0
	Augmented Type	Only abstract methods and three or more static final fields of the same type.	0-0	0-0
	Pseudo Class	Class which can be rewritten as an interface: no concrete methods, only static fields.	5-2	3-1
	Implementor	Concrete class, where all the methods override inherited abstract methods.	19-11	9-3
	Overrider	Class in which all methods override inherited, non-abstract methods.	27-20	11-14
	Extender	Class which extends the inherited protocol, without overriding any methods.	0-8	3-5
	Limited Self	Subclass that does not introduce new fields and all self method calls are to its superclass.	13-9	6-5
	Recursive	Class that has at least one field whose type is the same as that of the class.	7-11	3-7

Table 16: List of micro-patterns

Gil & Maman [62] classify micro-patterns as (a) Degenerate classes and interfaces which do not define any variable or methods, (b) Containment classes which explicitly manage their internal fields, or (c) Inheritance classes which inherit from other classes. The table includes counts of patterns used in all rules generated and in rules discovered in the SPEC benchmarks, as either sources or destinations. Rules were computed by using DaCapo benchmarks as the training set. In both cases, only rules with confidence greater than 75 are counted %. Micro-patterns never exploited (including those only discovered in classes compiled at build-time) are shown in italic.

Micro-patterns are similar to design patterns [60] except that micro-patterns stand at a lower level of abstraction, closer to the implementation. Gil and Maman define a micro-pattern to be "a non-trivial, formal condition on the attributes, types, name and body of a class and its components, which is mechanically recognisable, purposeful, prevalent and simple".

While design patterns focus on interaction between classes, micro-patterns define class-level properties. Each micro-pattern can be expressed as a formal condition on the structure of a class, so a class may exhibit several micro-patterns. A set of micro-patterns may implement a design pattern.

In addition, the authors show that different implementations of a same specification are likely to implement similar micro-patterns. Statistical analysis of a very large corpus, drawn from a wide variety of application domains, indicates that use of these patterns is not random. In Java, fields and methods can be:

- *Static*
- *Final*
- *Private*, *public*, *protected*, or default
- Inherited

On top of the above, methods can also be:

- *Abstract*
- Overriding
- Refining
- Constructor method
- Anonymous static initialiser

The properties above demonstrate the richness of the Java language, and Java classes can have numerous fields and methods exhibiting any combination of the above properties.

In their paper, the authors identified 30 different micro-patterns, which reflect the different characteristics of a class and its behaviour (see Table 16). Each micro-pattern describes a certain behaviour and state of a class.

Consider an example. *Sampler* is a 'controlled creation' micro-pattern[1]. It defines classes which have a *public* constructor and one or more *public static* fields of the same type as the class. Such classes provide clients with pre-made instances of the class as well as being able to make their own. *java.awt.Color*, which provides pre-defined colours, is a Sampler. The following code extract taken from the class *java.awt.Color* illustrates the above.

```
1  public class Color implements Paint, Serializable
2  {
3    public static final Color white = new Color(0xffffff, false);
4    public static final Color WHITE = white;
5
6    public static final Color lightGray=new Color(0xc0c0c0,false);
7    public static final Color LIGHT_GRAY = lightGray;
8
9    public static final Color gray = new Color(0x808080, false);
10   public static final Color GRAY = gray;
11   [...]
12   public Color(int red, int green, int blue){
13     this(red, green, blue, 255);
14   }
15
16   public Color(int red, int green, int blue, int alpha){
17     if ((red & 255) != red || (green & 255) != green ||
18           (blue & 255) != blue    || (alpha & 255) != alpha)
19           throw new IllegalArgumentException("Bad RGB values "
20           +"red=0x"+Integer.toHexString(red)
```

[1]'Controlled' creation differs from 'restricted' creation because the former defines a public constructor.

```
21            +"green=0x"+Integer.toHexString(green)
22            +"blue=0x"+Integer.toHexString(blue)
23            +"alpha=0x"+Integer.toHexString(alpha)    );
24
25      value = (alpha << 24) | (red << 16) | (green << 8) | blue;
26      falpha = 1;
27      cs = null;
28    }
29    [...]
30 }
```

As one would expect, sampler objects turn out to be very likely immortal.

Another example is *Joiner*, which is defined as "an empty interface joining two or more superinterfaces".

The catalogue they provide captures a wide spectrum of common coding practices, including particular uses of immutability, wrapping, restricted creation and emulation of different programming paradigms with object-oriented constructs.

6.2 Motivation

Often, programmers have an intuition about the lifetime of some objects. Yet, this intuition might be unreliable, because programmers make mistakes, but also because as code evolves, the lifetimes of objects might change. Determining object lifetimes by asking the programmer is therefore unreliable.

Programmers are often encouraged to follow coding style guidelines to ensure their code is as reliable and as reusable as possible. This is sometimes enforced by company policies, or learned through experience or education (courses, books, design patterns, etc.). The use of standard libraries might also require programmers to adopt a particular programming style. Although programs are written in different styles, programmers tend to adopt similar practices and idioms.

It was shown that micro-patterns can capture particular aspects of Java classes.

They are interesting for several reasons.

Firstly, in terms of level of abstraction, they are close to the implementation. Therefore, they are potentially capable of capturing common coding practices. The intuition is that by capturing common coding practices, it may be possible to discover a correlation between objects that belong to certain classes and lifetimes.

Secondly, Gil and Maman found that different implementations of the same specification are likely to implement similar micro-patterns. If different implementations of a same specification also have similar object lifetimes distributions, then there likely exists a correlation between micro-patterns and object lifetimes.

6.3 Methodology

The methodology employed is detailed in Chapter 3. A brief reminder of the way the experiments were conducted is presented below:

1. Object lifetimes were gathered from the training set (the DaCapo benchmarks).

2. Each allocation site was associated a lifetime, based on the lifetimes of the objects that were allocated at this site.

3. Source micro-patterns and destination micro-patterns were associated with allocation sites using Gil and Maman's tool.

4. Ar elation between sets of micro-pattern source and destinations and object lifetimes was then derived.

5. This relation is then data-mined and rules at various confidence levels are generated.

6. Finally, the experiment is run by loading the relevant advice into the JVM using the command-line parameter.

The following sections compare the results against self-prediction for both GC time and throughput. Note that the self-prediction results were gathered by loading a pre-generated self-prediction advice file into the experimental JVM. Also, note that in the experiments, results are compared against Jikes RVM version 2.4.4, which incorporates an immortal space by default, whereas Blackburn et al. [26, 22] compare their results against a version of Jikes RVM which did not incorporate an immortal space.

6.4 GC Time

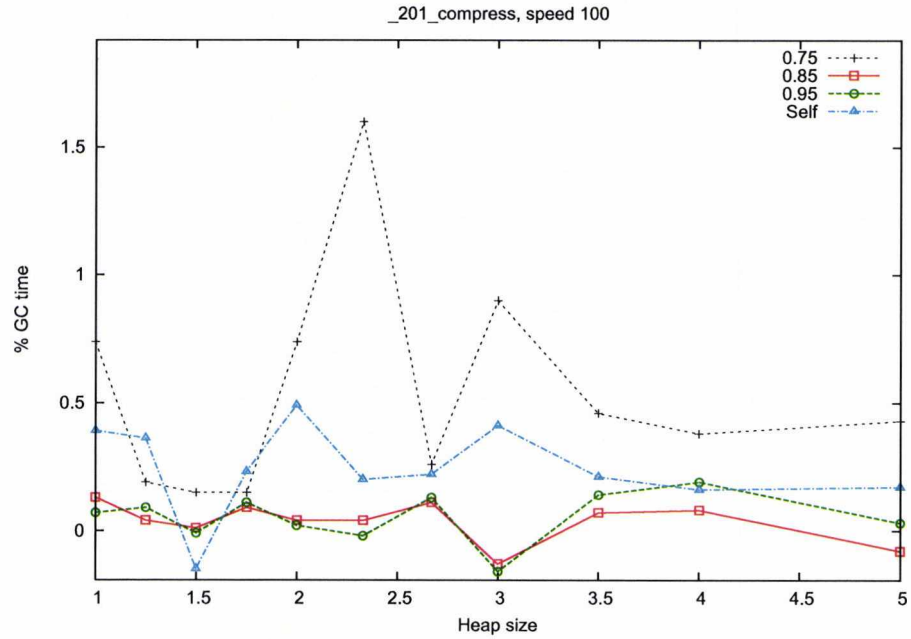
This section analyses the performance improvements in terms of GC time achieved by generating pretenuring advice, based on the set of source and destination micro-patterns of each allocation site. The performance of the system is analysed for each benchmark individually, before reviewing the overall performance improvements obtained using particular confidence levels.

6.4.1 Individual Benchmark Comparison

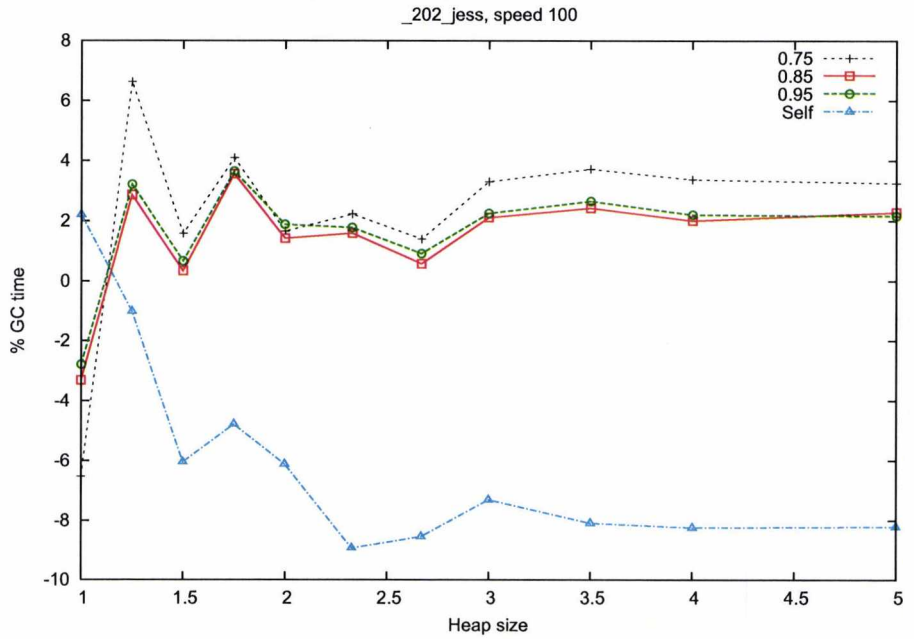
Each graph shows the relative time spent doing GC when compared with a run of the same program using a default JVM which does not implement the advice system (see Chapter 3). This paragraph analyses the performance of the system for each benchmark individually.

Figures 29, 30, 31 and 32 show relative GC time performance for the eight jvm98 benchmarks and contain two graphs each. In each figure, three curves show performance at various confidence levels (75%, 85% and 95%). The remaining curve shows the performance of self-prediction, the best possible scenario (see Chapter 3). In these Figures, a performance of +X% would mean that performance is degraded by X% while -X% would mean performance is increased by X% (hence spending X% less time in GC).

In Figure 29, the advice is neutral for compress in most cases, except at 75% confidence where it is slightly worse, with a GC time performance degradation

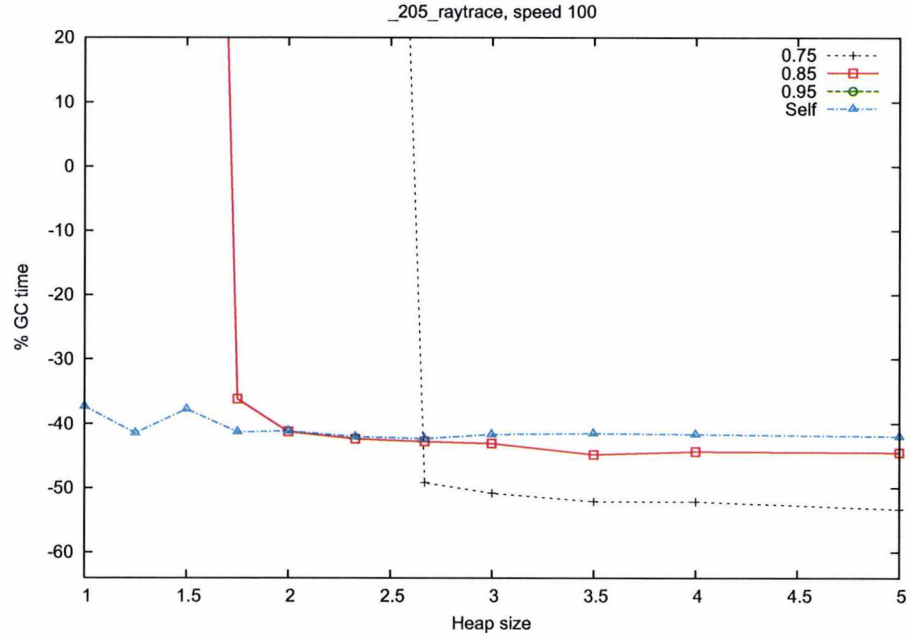


(a) compress

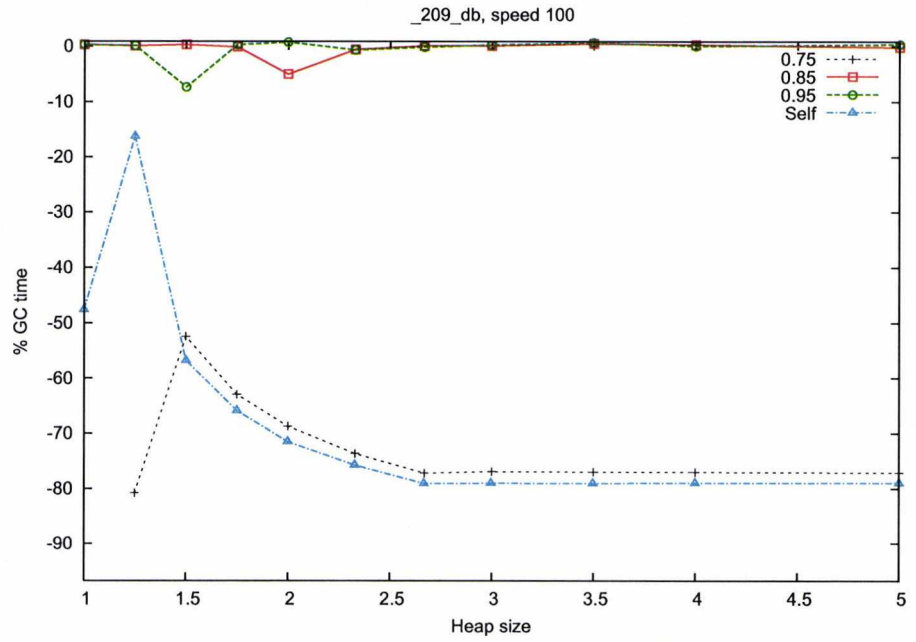


(b) jess

Figure 29: GC time relative to no advice for GenCopy configurations at 75, 85 and 95% confidence and for self-prediction.

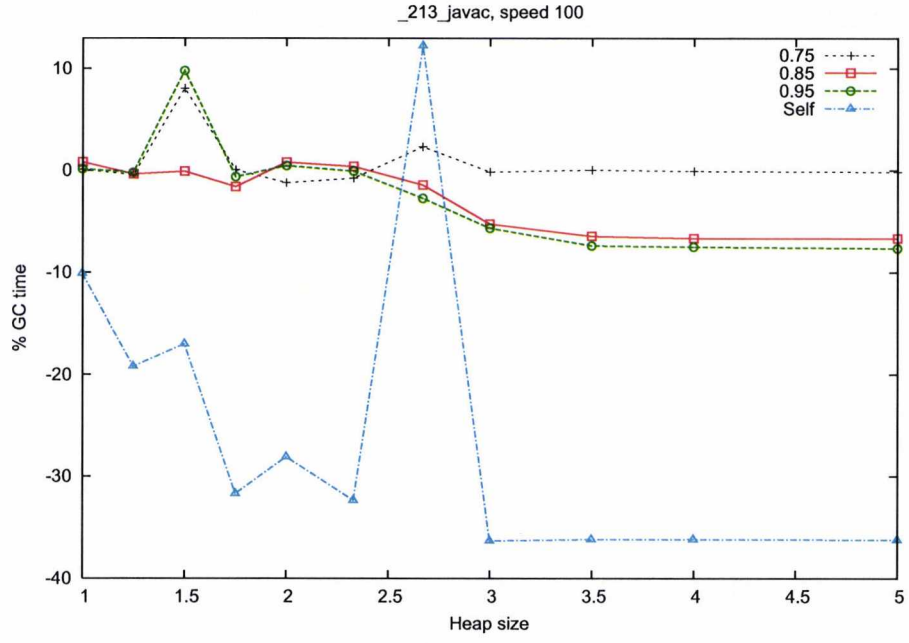


(a) raytrace

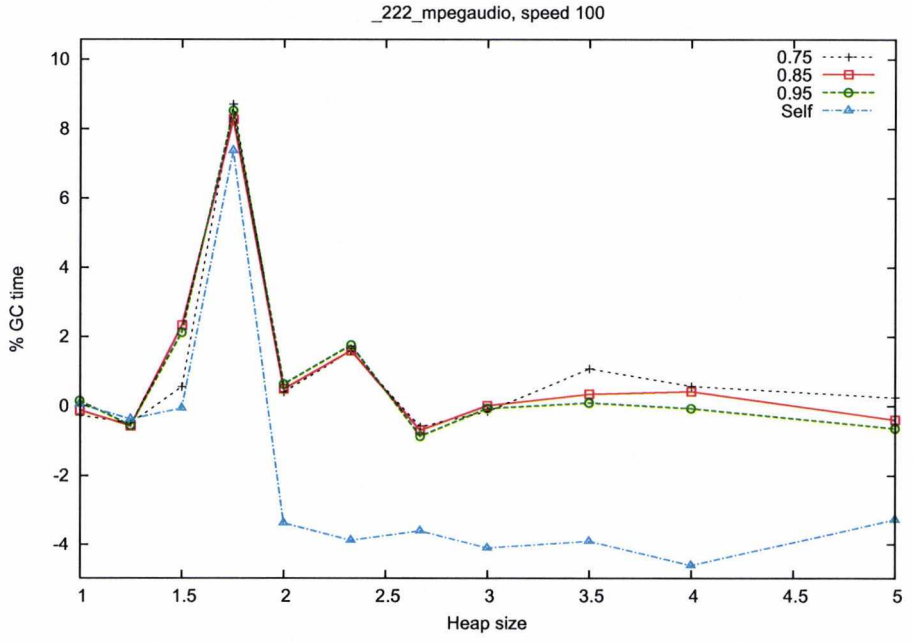


(b) db

Figure 30: GC time relative to no advice for GenCopy configurations at 75, 85 and 95% confidence and for self-prediction.

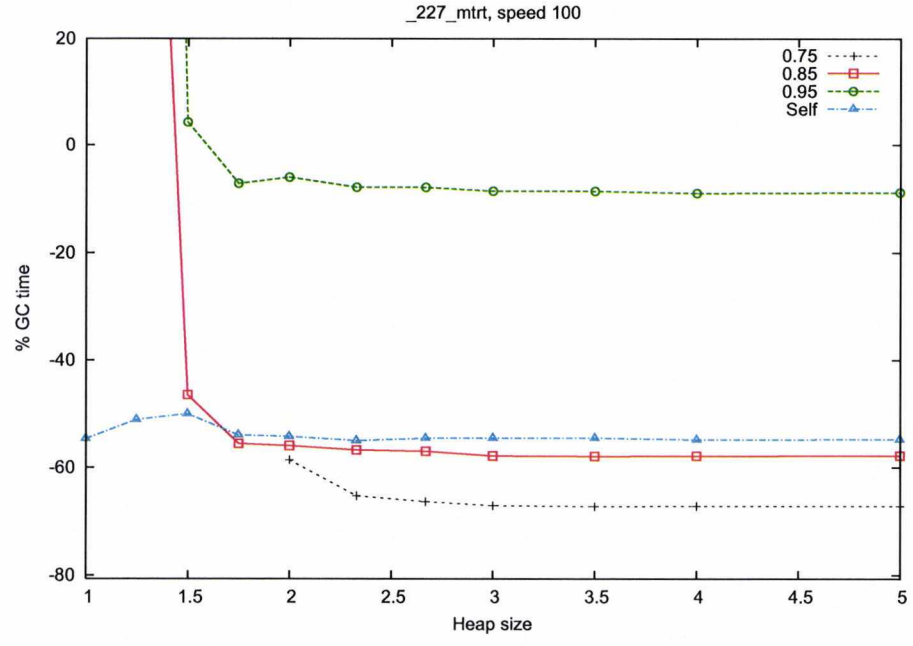


(a) javac

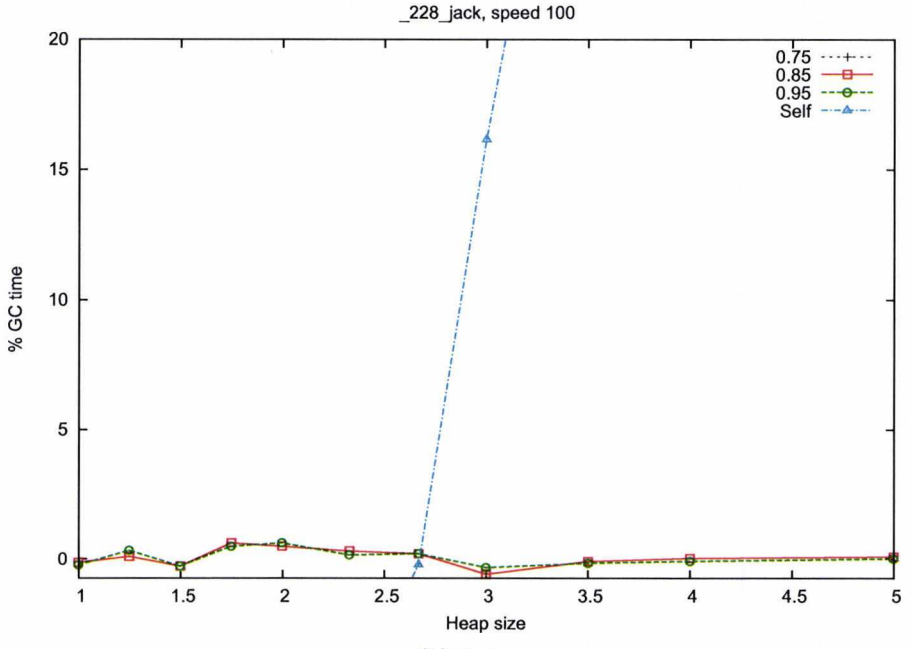


(b) mpegaudio

Figure 31: GC time relative to no advice for GenCopy configurations at 75, 85 and 95% confidence and for self-prediction.



(a) mtrt



(b) jack

Figure 32: GC time relative to no advice for GenCopy configurations at 75, 85 and 95% confidence and for self-prediction.

of up to 1.5%. Note that in this case, self-prediction does not produce any performance gain either. Jess shows performance improvements for self-prediction of up to 8.5% while the advice produces a performance degradation of 2% on average.

Figure 30 shows performance graphs for raytrace and db. In both instances, the advice performs extremely well, with performance improvements of up to 50% for raytrace and up to 78% for db. When the heap size is less than 2.5 times the minimum heap size, using the advice at 75% confidence, the JVM runs out of memory. However, when the heap size is more than 2.5 times the minimum heap size, the advice at 75% confidence performs closely to self-prediction for db, and even improves over self-prediction for raytrace. This behaviour is explained by the fact that 75% confidence is an aggressive confidence level, which in this case advises pretenuring a large number of objects. If the heap is large enough, it improves performance, but if the heap is too small, the JVM runs out of memory due to over-pretenuring. In db, at very tight heap sizes (1.25 times minimum), the advice at 75% confidence performs better than self-prediction, and follows the curve of self-prediction thereafter by 1 to 2%.

In Figure 31, performance improvements can be seen for javac of up to 7% at large heap size when using a confidence level of 85% or above. However, mpegaudio is mostly unaffected by the advice except for a performance spike at 1.75 times the minimal heap size, which is also noticed for self-prediction.

Finally, Figure 32 shows significant performance improvements for mtrt, which is a variant of raytrace, of up to 64% for a prediction level of 75%. For mtrt and at large heap sizes, the advice is also able to beat self-prediction. Regarding jack, the advice offers no performance improvements and remains neutral, except at 75% confidence, where performance is degraded by a very large factor (the graph does not show the degradation as the performance degradation is too high) due to over-pretenuring. Note however that jack's performance is also degraded when using self-prediction if the heap is larger than 2.5 times the minimum heap size.

6.4.2 Summary

In summary, 75% confidence generally offers the largest performance gains, improving raytrace and mtrt by around 50% and db by 65% on average. However, in tight heaps 75% confidence pretenures too much, causing raytrace and mtrt to run out of memory. Javac in contrast only shows improvements (around 7%) for confidences $\geq$ 75%, in large heaps.

Figure 33 shows a summary of the prediction efficiency at 75% confidence. 75% confidence gives good GC improvement for the three benchmarks raytrace, mtrt and db. One benchmark however (jack) performs very poorly due to wrong predictions and does not appear on the figure, while other benchmarks are mostly neutral.

Figure 34 shows a summary of the prediction efficiency at 85% confidence. In this graph, given a large enough heap ($>$ 1.75 times the minimum heap size), the advice improves greatly 2 benchmarks (mtrt and raytrace) by up to 58% while being neutral with regards to GC performance for other benchmarks.

Figure 35 shows a summary of the prediction efficiency at 95% confidence. In this graph, performance gains are modest and 2 benchmarks (javac and mtrt) are improved upon. Excluding tight heaps ($<$ 2 times the minimum heap size), the advice degrades performance in only one case (jess) and by only up to 3%.

6.5 Overall execution time

Figures 36, 37, 38 and 39 show overall execution times relative to no advice for a range of confidences. At 75% confidence, the gains in GC performance are generally not reflected in overall execution time. Given that GC time represents only a small fraction of overall execution time for the jvm98 benchmarks (see Figure 42), it is not surprising that changes in GC time lead to much smaller changes in execution time. However, micro-pattern advice offers performance similar to self-prediction for raytrace and mtrt at 85%, and for javac at 75% in large heaps.

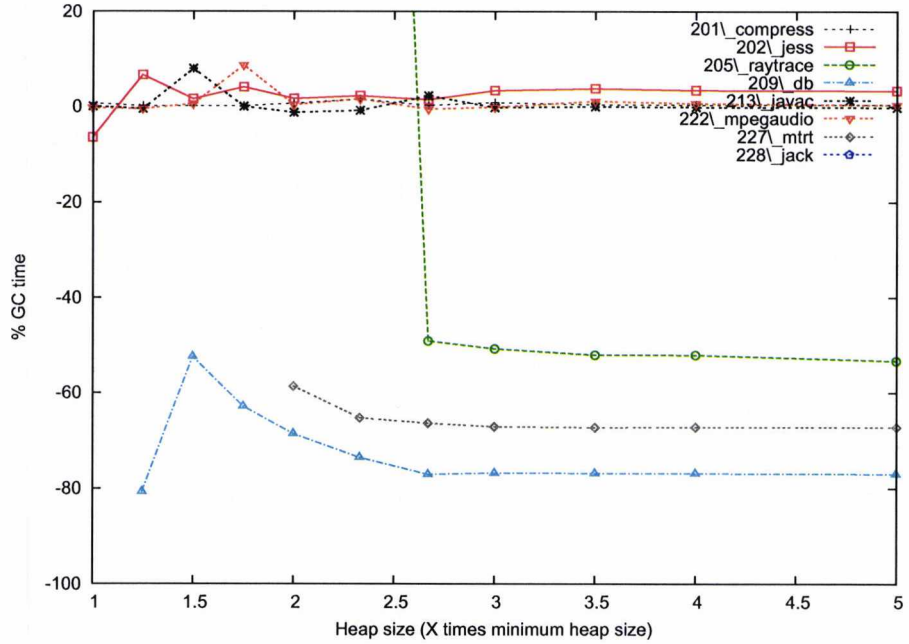


Figure 33: GC time relative to no advice for GenCopy configurations at 75% confidence and for self-prediction.

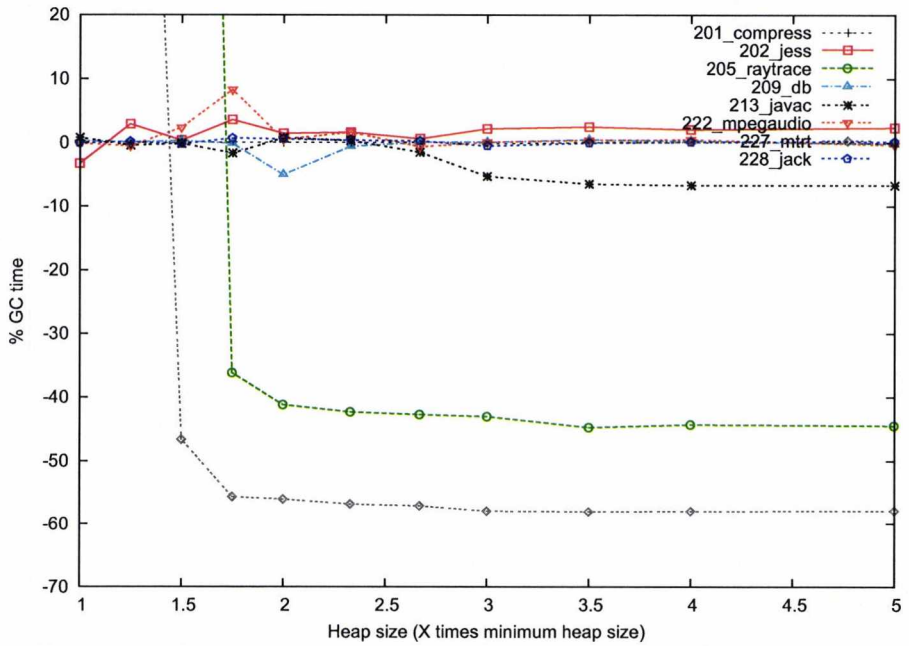


Figure 34: GC time relative to no advice for GenCopy configurations at 85% confidence and for self-prediction.

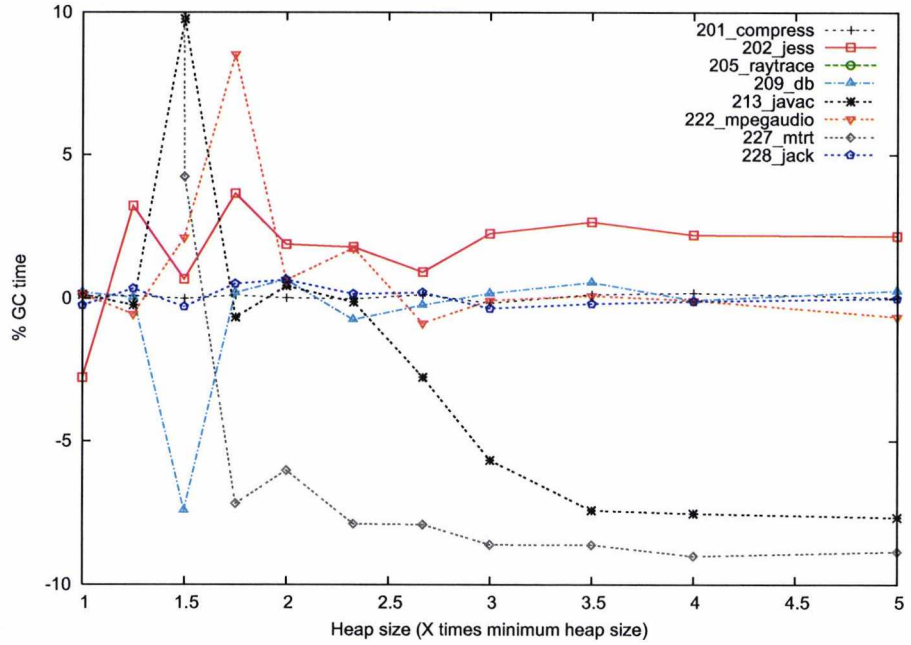


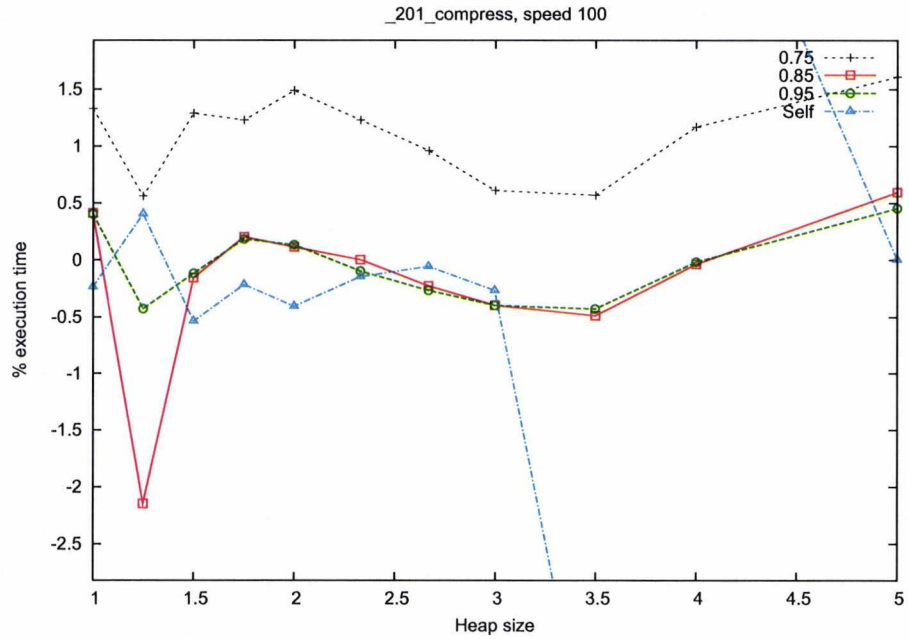
Figure 35: GC time relative to no advice for GenCopy configurations at 95% confidence and for self-prediction.

But, some results are counter-intuitive. For example for db, 75% confidence pretenuring improves GC time by 65% but gives extremely poor performance overall. The reasons for this behaviour are explored in Section 6.7.

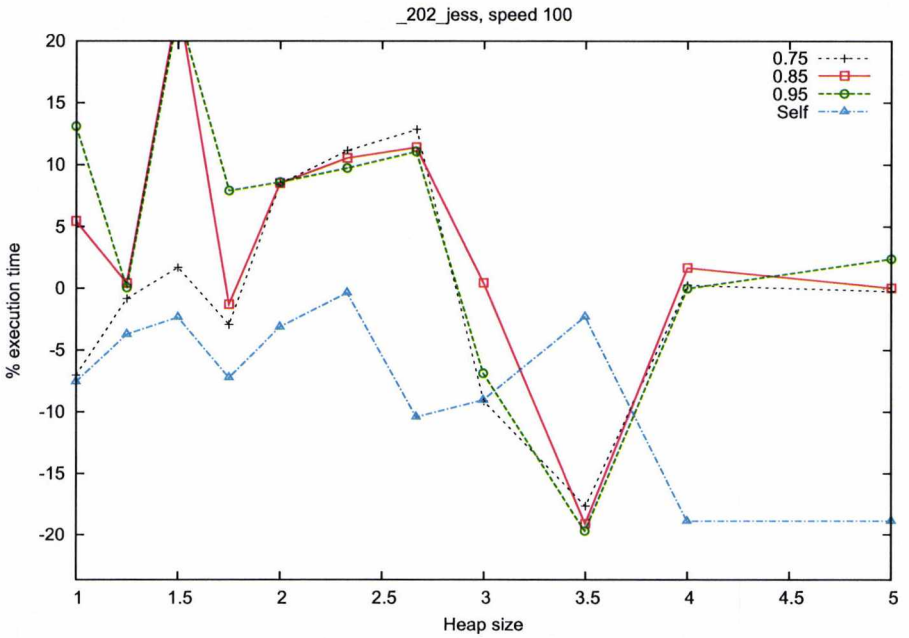
6.6 GC pause time

Pretenuring aims to reduce the volume of data copied at each collection. Because a copy-reserve is not needed when pretenuring immortal objects, the effective heap can be increased, thereby giving objects more time to die and reducing the volume of objects to be copied. Since less data should be copied, one could anticipate that the scheme should reduce pause times. For minor collections, pretenuring increases the proportion of short-lived objects allocated in the nursery which again should reduce pause times.

Figures 40 and 41 show the pause time distributions of several benchmarks at specific configurations where the system improves overall GC time. Configurations where the system performs well in terms of GC time we purposefully chosen in

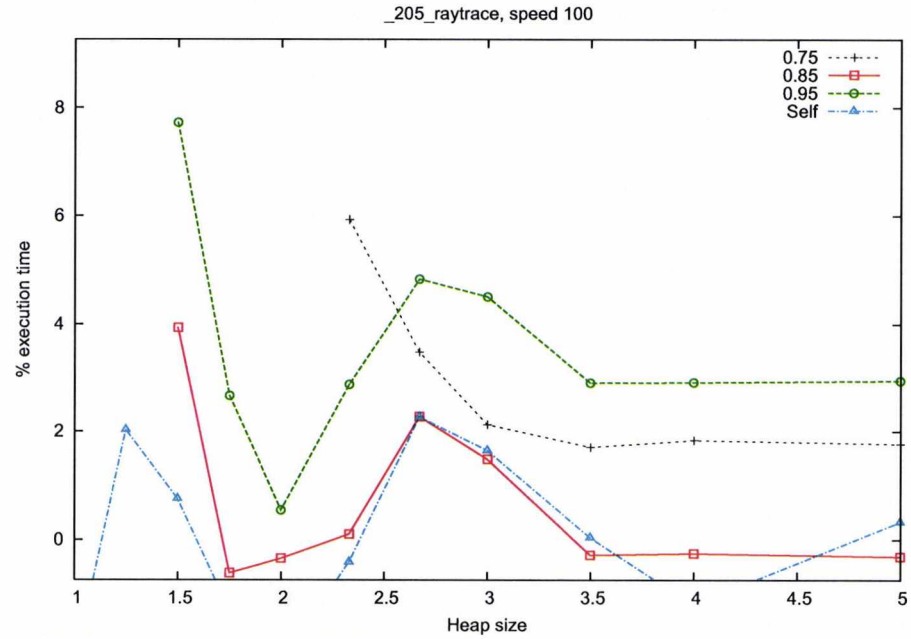


(a) compress

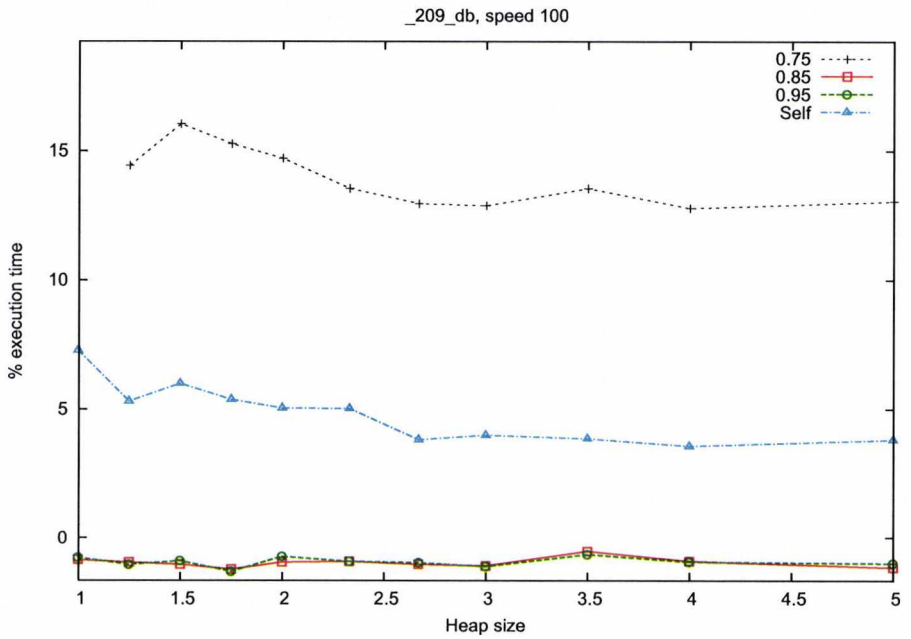


(b) jess

Figure 36: Overall execution times relative to no advice.

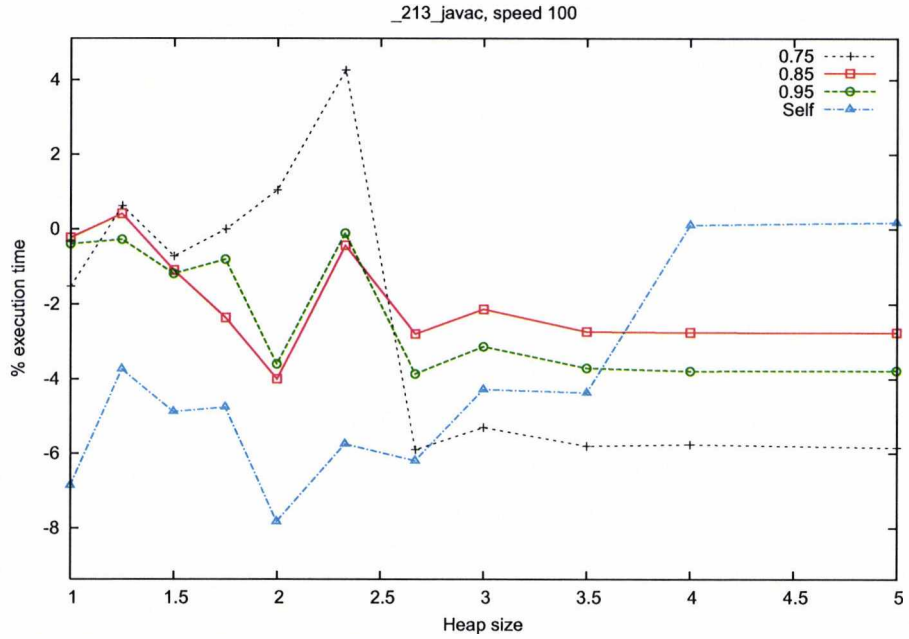


(a) raytrace

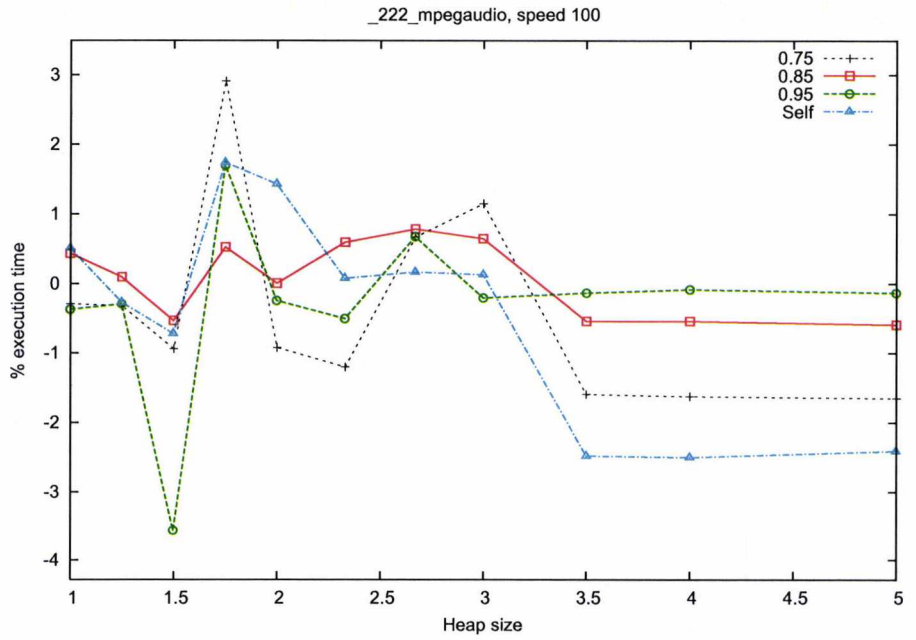


(b) db

Figure 37: Overall execution times relative to no advice.

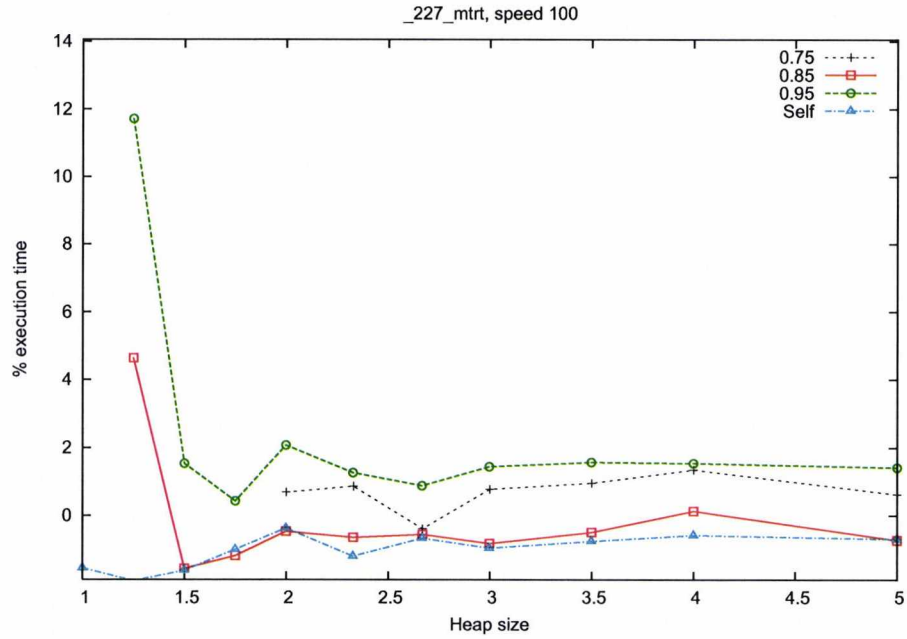


(a) javac

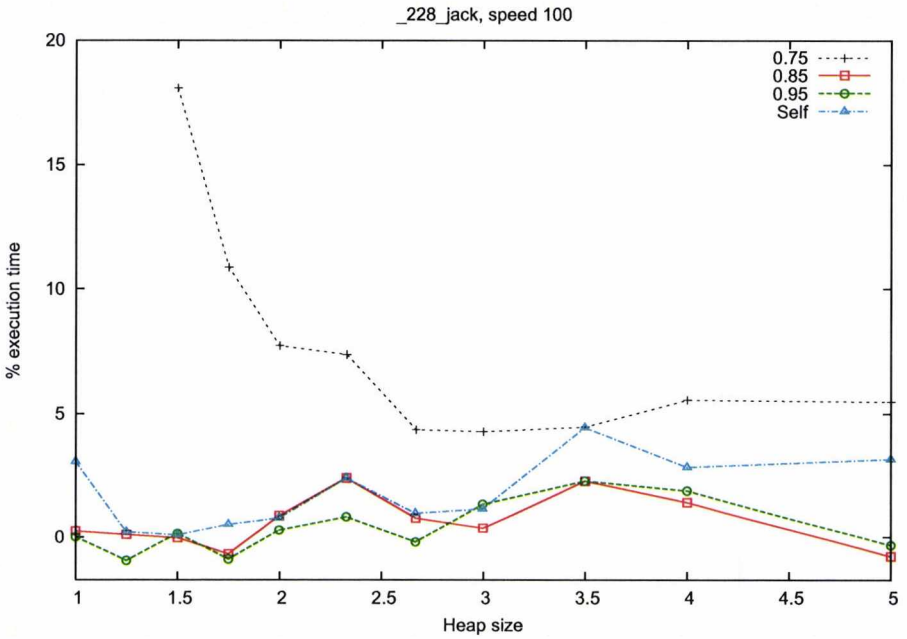


(b) mpegaudio

Figure 38: Overall execution times relative to no advice.

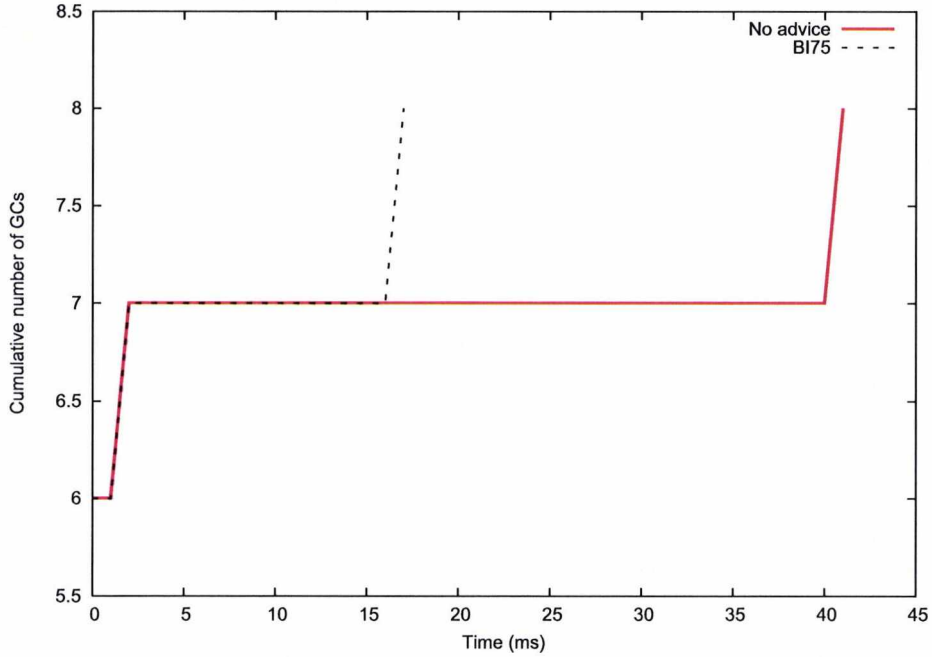


(a) mtrt

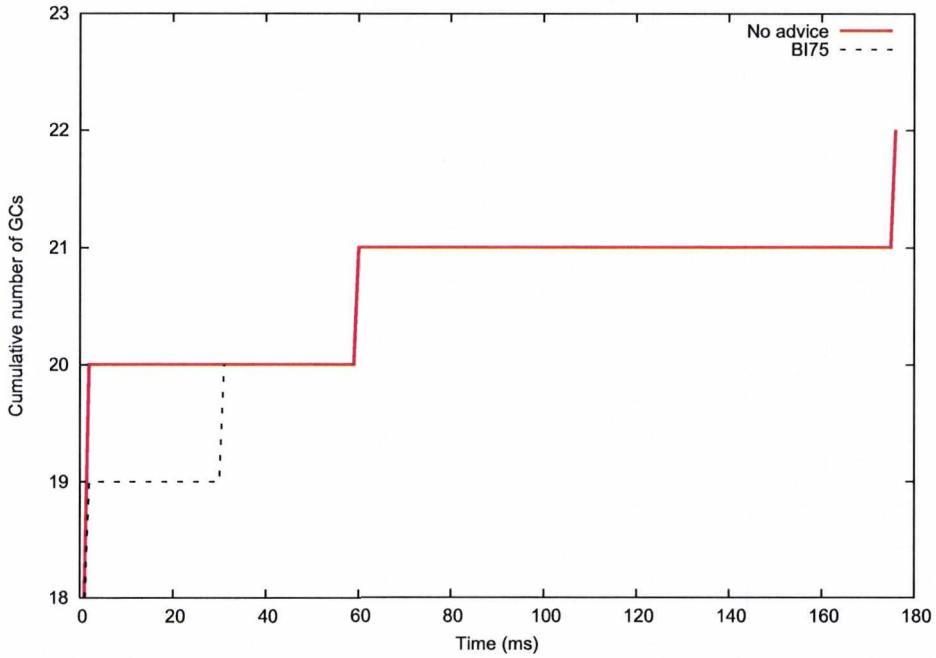


(b) jack

Figure 39: Overall execution times relative to no advice.

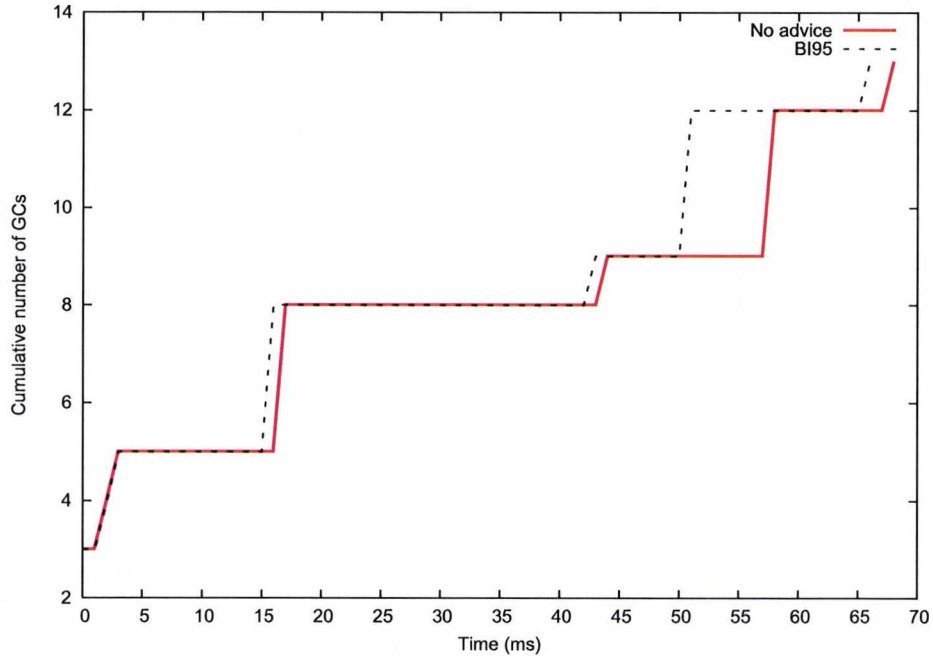


(a) raytrace, 3×, conf. 75%

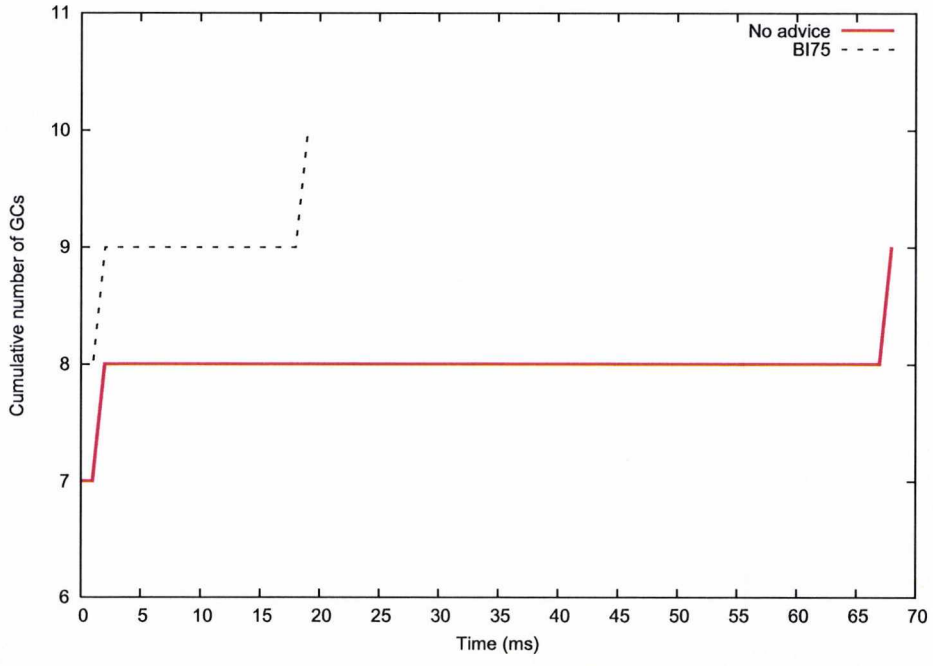


(b) db, 1.25×, conf. 75%

Figure 40: Cumulative pause time distributions, compared with no advice. A point (x, y) on the line indicates that y collections had a pause time less than x ms.



(a) javac, 3.5×, conf. 95%



(b) mtrt, 2.33×, conf. 75%

Figure 41: Cumulative pause time distributions, compared with no advice. A point (x, y) on the curve indicates that y collections had a pause time less than x ms.

order to demonstrate the potential pause-time savings that this approach can offer.

The Y axis shows the cumulative number of GCs compared with the cumulative time in which GCs were performed (X axis). Lower curves indicate that fewer GCs were triggered, and more-left curves indicate that the pauses were shorter.

Each graph compares a run of a non-modified JVM (*no advice*) with a run of the experimental JVM loading an advice file. In these graphs, *BI75* means that Blackburn et al.'s version of immortal objects (see Section 2.4.7) is used at 75% confidence. Likewise, *BI95* means that Blackburn et al.'s version of immortal objects is used at 95% confidence.

Figure 40 (a) shows the pause-time distribution for raytrace, at 3 times the minimum heap size and using a confidence level of 75%. As can be seen, the same number of GCs were triggered in both cases, but the default system spent 41 ms performing GC, while the advice system spent 17 ms.

Figure 40 (b) shows the pause-time distribution for db, at 1.25 times the minimum heap size and using a confidence level of 75%. Here, the default JVM performed a total of 22 GCs, and spent 176 ms performing GCs, the longest pause being of 116 ms. In comparison, the system required only 20 GCs, and spent a total of only 31 ms doing GC.

Figure 41 (a) shows the pause-time distribution for javac, at 3.5 times the minimum heap size and using a confidence level of 95%. Here, 13 GCs were triggered in each case, but the advice curve is shifted left with the advice system, indicating that the system spent less time in GC than the default JVM. The pauses were in general shorter with the advice system than the default JVM, with the exception of the 12th GC, which was longer using the advice system. In both cases, the longest pause was 27 ms.

Finally, Figure 41 (b) shows the pause-time distribution for mtrt, at $2\frac{1}{3}$ times the minimum heap size and using a confidence level of 75%. In this case, the default system spent a total of 68 ms performing GC, the longest pause being 67 ms. In comparison, the system spent only 19 ms in GC, the longest pause being

of 16 ms.

This study demonstrates that in the cases where overall GC time is improved, pause time is also reduced for both full heap collections (although jvm98 benchmarks do very few full collections) and minor collections. In the case of db at 1.25 times the minimum heap size and using a confidence level of 75%, the time spent in GC was reduced by a dramatic 546%, while the longest pause was reduced by 400%.

6.7 Performance Considerations

When considering pretenuring, there are several implications one has to bear in mind. First, objects allocated in the immortal space must have a bit set to the current value of the marking bit (see Section 4.6.2). Second, although pretenuring increases the space used by the immortal region, this frees copy reserve, thereby increasing the effective heap available to other spaces, provided the pretenuring decision is correct and does not increase the volume of floating garbage. Pretenuring at 75% confidence places no objects in the mature space (although pretenuring at more aggressive, lower confidence levels does) but places many objects in the immortal space. Third, increasing the volume of objects in the immortal and mature spaces might change the number of cross-region references which have to be trapped by the write-barrier's slow path and added to the remembered set (see Section 6.7.1).

6.7.1 db's Behaviour Reviewed

Unfortunately, improvements in GC time do not always translate into improvements in execution time.

When comparing GC time performance with execution time performance, db at 75% confidence shows the most disparity by improving GC time and degrading overall throughput. The causes of the extra mutator overhead are explored below.

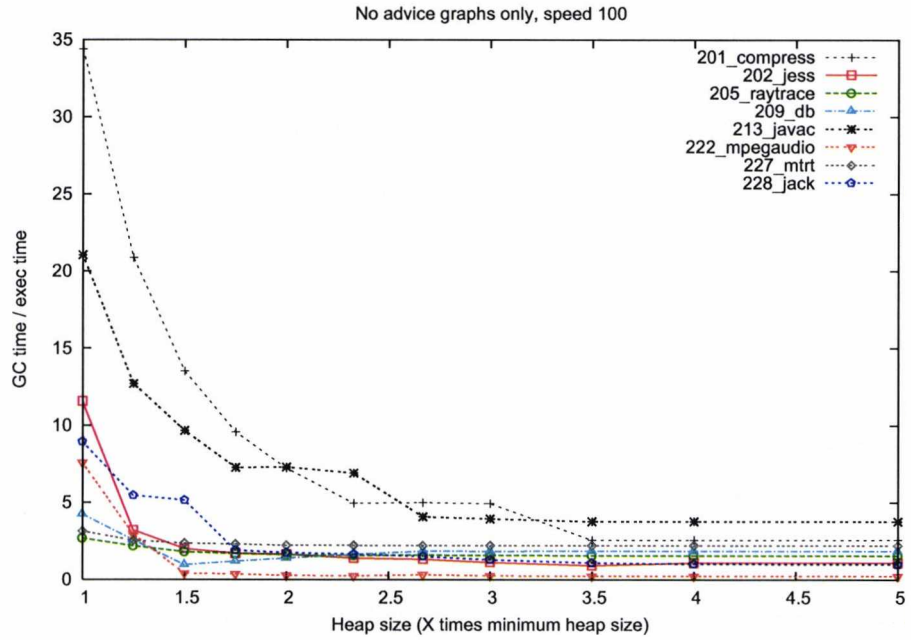


Figure 42: GC time as a fraction of overall execution time for MMTk GenCopy (without advice).

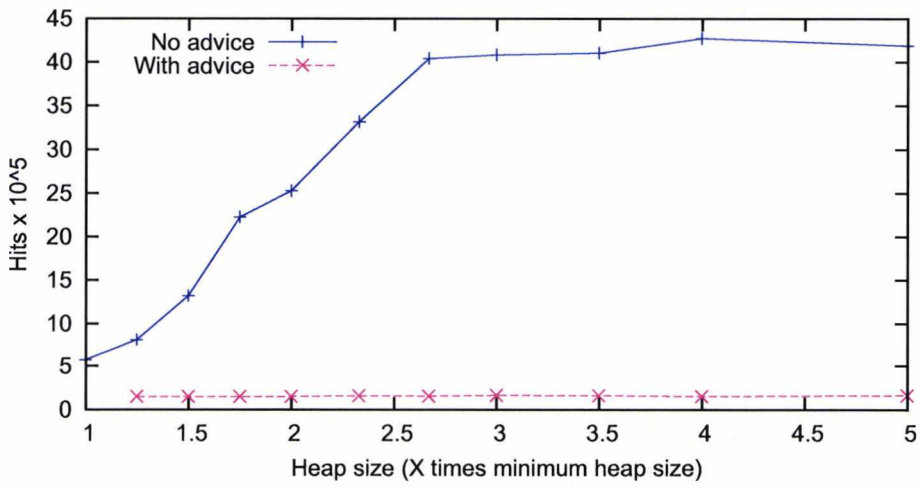


Figure 43: The number of times that the write barrier's slow path is taken, with and without advice, by db at 75% confidence.

Cost Of Setting Bits

As explained above, a bit needs to be set to the current value of the marking bit every time an immortal object is allocated. To estimate a lower bound on the additional cost of allocating objects in the immortal region, the cost of setting a header bit at *all* allocations was measured and the fraction of this cost proportional to the number of immortal allocations was calculated.

This is a lower bound because it can be assumed that the value of the bit will stick in the cache, which it might not if bit-setting is unusual (as it would be for immortal allocation). Under this pretenuring scheme db allocates 40,047 objects in the immortal space. However, the performance results obtained show negligible throughput variation between performing bit-setting at each allocation and running the system without advice.

Write-Barrier Slow Path

Every-time a pointer is written to point to an object located in a different region, which can be independently collected, the write-barrier slow path is taken in order to update the remembered sets (see Chapter 2.1.3). This operation is costly.

Figure 43 shows the number of times that the write-barrier's slow path is taken, with and without advice, by db at 75% confidence. As can be seen, for *no-advice*, the larger the heap size, the more frequently the write-barrier slow path has to be taken. This is because in an Appel-style collector [3], the larger the nursery is, the fewer collections are required, and the later the object is promoted. Delaying the promotion of objects increases the number of potential old-to-young pointers, and hence increases the number of times the write-barrier has to be taken.

However, using the advice, objects advised to be pretenured are immediately allocated together in the same space. Therefore, rather than increasing the number of times the write-barrier slow path is taken, pretenuring substantially reduces it.

Heap Size	1.5 X	1.75 X	2 X
No Advice	244889997	247169698	247057737
Advice	326828354	328105942	328808123
% variation	+33.5%	+32.7%	+33.1%

Table 17: Number of DTLB cache misses for _209_db at various heap sizes for advice and no advice.
Heap sizes are expressed as multiples of the minimum heap size. *No Advice* and *Advice* show the number of DTLB cache misses for a system running without advice, and a system running with advice respectively.

Layout Of Data

Pretenuring can disrupt the layout of data by allocating objects that would otherwise be allocated together in different regions. In the case of db, this is of prime importance since db is sensitive to the layout of data as it repeatedly traverses long singly-linked lists [22].

Examination of the hardware performance counters in table 17 indicates that, although the number of instructions executed and the cache behaviour for the mutator is similar with and without advice, advice increases DTLB misses by 33%.

Finally, as can be seen in figure 42, it is worth mentioning that GC time accounts for only a very small proportion of overall execution time, so even small mutator overheads will override any improvements in GC time.

6.8 Analysis

In MMTk, objects that are known to be immortal are allocated directly in the immortal space. Table 18 summarises the consequences of the pretenuring decisions made by MMTK as well as the consequences of using the advice at 75% confidence.

The third column of Table 18 shows the number of sites in the jvm98 suite for which predictions with confidence greater than 75% are obtained. At this confidence level (and above), there are no predictions of long-lived objects, although

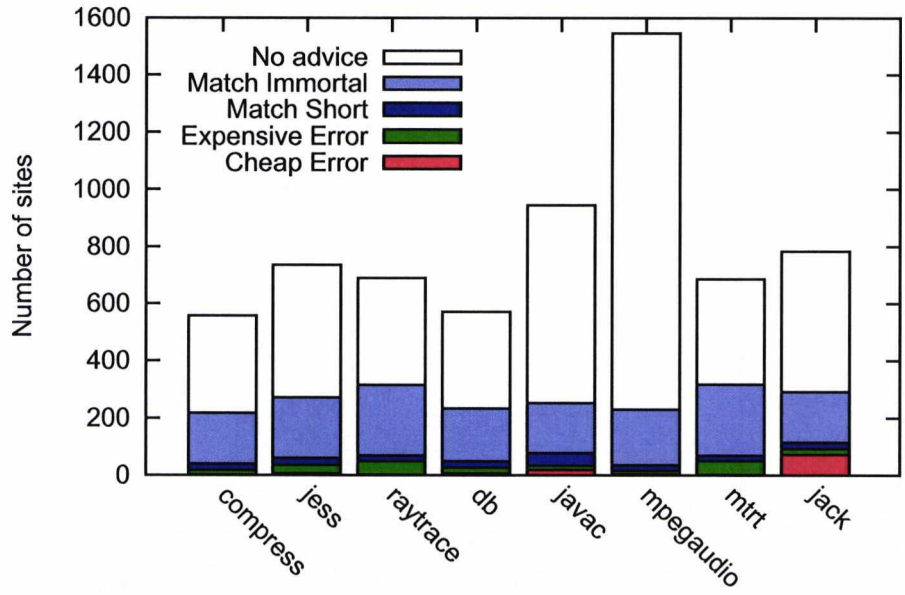


Figure 44: Comparing self prediction with micro-pattern advice at 75%.

these appear at lower confidence levels. Most predictions are of immortal sites.

Amongst the 97 rules at 75% confidence or above generated by the data-mining tool, less than half are used in jvm98. Figure 45 shows the number of times each rule is used in each benchmark. Most rules apply to only a few sites, but some rules apply to many sites. Furthermore, most rules are used the same number of times in each benchmark, and different benchmarks are distinguished by use of only a very small number of rules.

Figure 45 is interesting as it shows that some rules are responsible for a large amount of pretenuring. Consider the case of raytrace and mtrt, two benchmarks in which the system dramatically reduces GC time at 75% confidence. Rule 25 in Figure 45 is more prevalent (with 72 instances) in these benchmarks than in other benchmarks. The rule states that the source is *Immutable* while the destination is *Immutable* and not a *CompoundBox*. This rule applies to sites which provide graphical components of the scenes to be raytraced: the **Scene** (the source) and objects within it (the destinations) are never changed, and common components like **Points** have only primitive fields.

	Min.	All		Loading overhead			Advice sites		Count		Volume		
Program	heap	sites	Rules	time%	before	after	Advice	MMTk	Advice	MMTk	Advice	MMTk	All
compress	21	6655	230	-0.55	8	8	203	83	1467	369	139	114	128
jess	22	6935	289	0.60	8	8	255	83	14871	647	535	133	299
raytrace	30	6757	328	0.63	8	8	303	83	2491889	416	54194	118	166
db	39	6665	246	0.13	8	6	215	83	140067	377	23784	114	101
javac	40	7269	271	1.16	16	13	204	83	25786	665	559	149	225
mpegaudio	18	7656	243	0.66	8	8	217	83	1463	461	140	120	44
mtrt	38	6756	331	0.60	8	8	306	83	2656466	419	57548	118	176
jack	22	6954	308	0.48	8	6	209	83	573176	469	13255	122	312

Table 18: Pretenuring placement at 75% confidence, all benchmarks with speed 100 input.
Min. heap is the size in MB of the smallest heap in which the program would run. *All sites* is the total number of sites compiled (including Jikes RVM, application and library code used). *Rules* is the number of sites with advice other than short-lived. *Loading overhead* is the percentage execution time overhead to load the advice file, and the space overhead (in pages) before and after the run. *Advice sites* is the number of sites pretenured to the mature space or the immortal space, either by advice or by MMTk; similarly, *Volume* and *Count* are the number and volume (in KB, or MB for *All*) of objects pretenured.

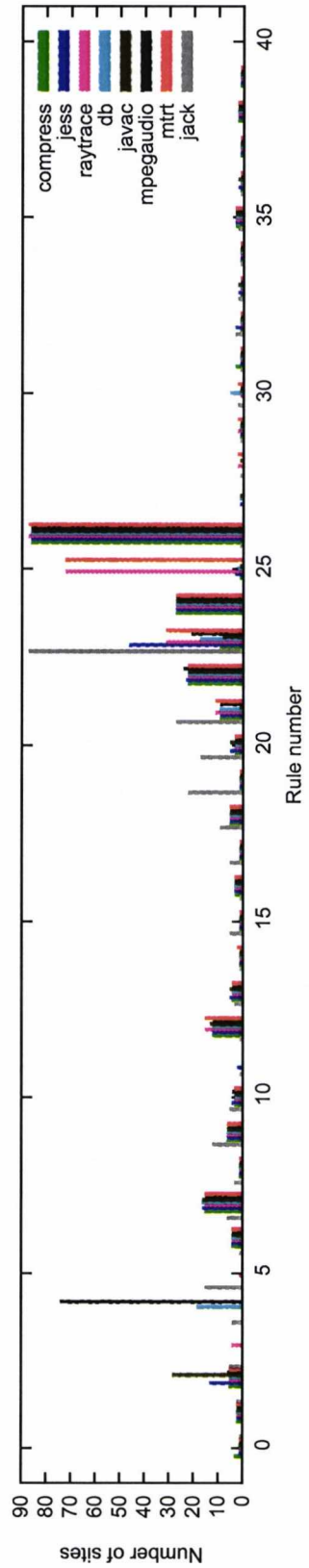


Figure 45: Instances of rules used by SPEC jvm98 sites at confidence 75%.

Figure 44 compares the accuracy of the micro-pattern predictions at 75% against self-prediction. The height of each bar is the total number of sites that allocated data, and each bar corresponds to a specific benchmark. White blocks illustrate the sites that were not predicted. Light blue and dark blue blocks are sites that the system predicted accurately (immortal sites and short-lived sites respectively). Green blocks are sites misclassified as immortal (*Expensive Error*), and red blocks are sites misclassified as short-lived (*Cheap Error*).

This graph shows that although the micro-pattern predictions are fewer than self-prediction, on average they match self-prediction advice for 81% of the sites for which they give advice. The prediction system makes few *expensive* errors (9% on average).

6.9 Statistical Significance

The previous sections of this chapter highlighted some significant improvements when using the methodology proposed in this thesis over a default system. While this approach offers important savings in GC time, it is important to statistically validate the results to establish that there really is a difference between self-prediction and the approach advocated in this thesis. The goal is to ensure these results were not obtained by pure luck.

To this aim, the primary objective of the analysis is to determine if there is any difference between:

- Self-prediction and micro-patterns at confidence 75% (Self-MP75).
- Self-prediction and micro-patterns at confidence 95% (Self-MP75).

A Sign Rank test can be used in replacement of a Student's T test when the normality assumption of the data is not met. The nonparametric Sign Rank does not require specific distributional assumptions about the data. When data is not normal, the parametric Student's T-test may not be robust, and the efficiency or power of the test may be compromised. Whereas when the data is normal,

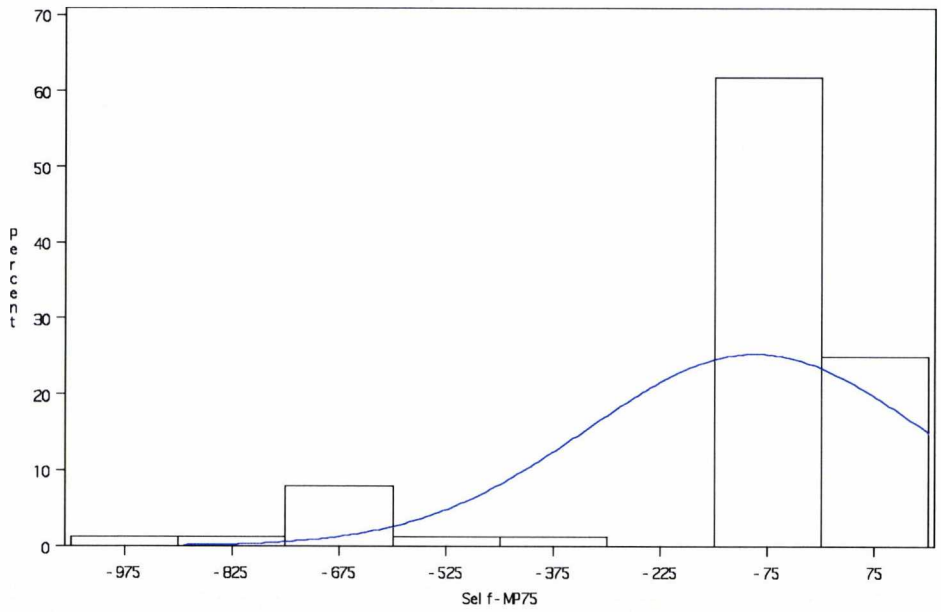


Figure 46: Plot of histogram for Self-MP75.

loss of efficiency or power may occurr with a nonparametric test. Therefore, it is important to check the distribution of the data prior to analysis. Also to be noted, here we assume that program is not a group variable of interest. Hence, the data is pooled across all 8 programs.

The histogram plots, normality probability plots and the normality tests can provide information about the distribution of the data.

Figure 46 shows the data for variable Self-MP75 (difference between Self and MP75) is left skewed. The same data distribution also applies for variable Self-MP95, see Figure 47.

The normality tests and the normal probability plots also suggest that the data for both Self-MP75, and Self-MP95 are non-normal. At 5% significance, the p-values for the 4 different variations of normality tests are all less than 0.05 (see Table 19 and 20). Therefore, we have evidence against the null hypothesis that the data is normal and Sign Rank test is the preferred test.

At 5% significance, the p-values are less than 0.05 (see Table 21). Therefore, we have evidence against the null hypothesis that the median is equal between

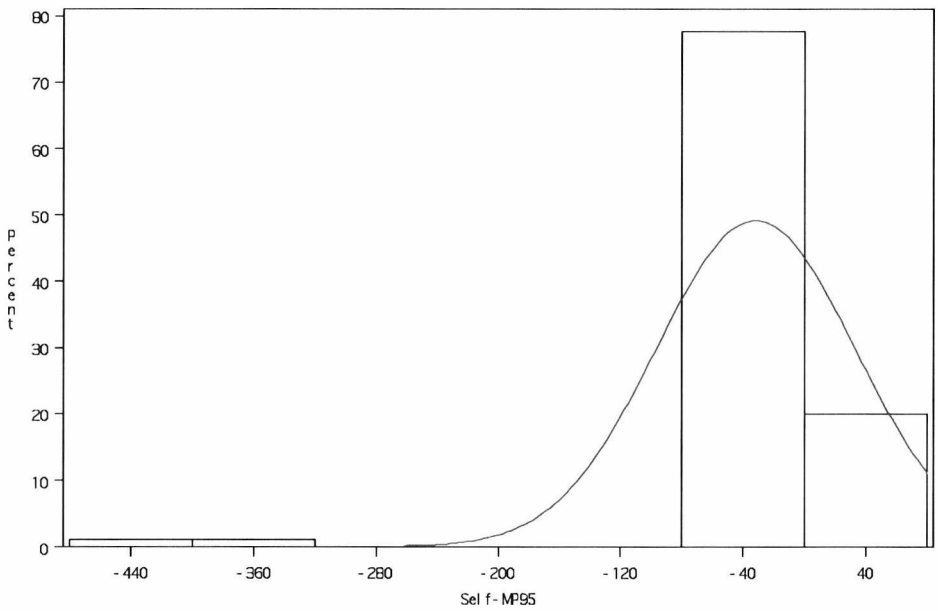


Figure 47: Plot of histogram for Self-MP95.

Test	P-value
Shapiro-Wilk	<0.0001
Kolmogorov-Smirnov	<0.0100
Cramer-von Mises	<0.0050
Anderson-Darling	<0.0050

Table 19: Normality Tests for Self-MP75.

Test	P-value
Shapiro-Wilk	<0.0001
Kolmogorov-Smirnov	<0.0100
Cramer-von Mises	<0.0050
Anderson-Darling	<0.0050

Table 20: Normality Tests for Self-MP95.

Variable	Sign Rank P-value	Student's T P-value
Self-MP75	<0.0001	<0.0012
Self-MP95	<0.0001	<0.0001

Table 21: Sign Rank and Student's T Tests.

- Self and MP75 (Sign Rank P-value <0.0001).
- Self and MP95 (Sign Rank P-value <0.0001).

Even though the Student's T test is not the preferred test here, it is worth noting that the Student's T P-values also suggest similar conclusions, see Table 21.

Therefore, we can conclude that there is a difference between self-prediction and the advice system at 75% confidence. There is also a difference between self-prediction and the advice system at 95% confidence.

6.10 Conclusion

Pretenuring long-lived and immortal objects into regions that are infrequently or never collected can reduce garbage collection costs significantly. However, extant approaches either require extremely computationally expensive, application-specific, off-line profiling, or consider only allocation sites common to all programs, i.e. those invoked by the virtual machine rather than application programs.

By data mining a large corpus of Java programs, it was possible to find relationships between micro-patterns exhibited at an allocation site and the lifetimes of the objects allocated by that site. This analysis is effective at discovering short-lived and immortal objects, but predicts fewer sites that allocate long-lived data. The system is capable of predicting the lifetime of objects in programs never encountered before within minutes.

Advice is loaded into the JVM at the start of the program, and the program can benefit immediately from the pretenuring decisions. Section 6.8 showed that the rules generated at 75% confidence matched self-prediction in 81% of the cases on average.

This analysis is cheap and could be provided in a class loader (though currently advice is loaded from files prepared off-line). Performance gains between 6 and 77% in GC time were obtained against a generational copying collector for several

jvm98 programs.

As demonstrated in Section 6.6, pretenuring decisions allow a significant reduction in pause-times, by up to 546%.

Unlike software metrics, which provide an understanding of relationships between classes, micro-patterns [62] offer an insight on the structure of each class. This lower-level view allowed us to capture programmer's intentions, mechanically categorise program classes and apply this classification to allocation sites. Furthermore, the data-mining classification process is much easier using micro-patterns than software metrics because micro-patterns are binary: a class exhibits a micro-pattern or it does not.

An example of the system predicting the lifetime of objects was shown in raytrace and mtrt which provide graphical components of scenes to be raytraced. This example showed that certain components within a program have particular lifetime characteristics. The system successfully captures some of it. With a larger training set, it is believed that the lifetimes of more object can be predicted, which may lead to more important and more consistent speedups across programs[2].

The next chapter presents the conclusions and future work.

[2]While is was not possible to acquire a larger training set, training on numerous real-life programs may lead to the discovery of more quality rules.

Chapter 7

Conclusions

This chapter presents the conclusions of this thesis and plans for future work. Section 7.1 of this chapter summarises the novel approach to pretenuring explored in this thesis and the results this approach led to. Section 7.2 discusses the reasons behind the poor results obtained using software metrics, compared to the convincing results obtained using micro-patterns. Finally, Section 7.3 presents plans for future work. In particular, this last section discusses enhancements that could be made to this approach, what other predictors could be explored, and what other types of garbage collector could be used.

7.1 Summary of the Thesis

This thesis has presented a study of how predictors based on static class properties can be used to efficiently predict the lifetime of objects in memory. This research has shown very good results when using the right kind of predictors, and opportunities- for future work are promising.

The remainder of this sections discusses the main contributions of this thesis.

7.1.1 State Of The Art Pretenuring

By allocating long-lived objects directly into longer-lived generations, pretenuring can reduce the amount of copying work the garbage collector has to perform at each collection [26, 22] . By allocating immortal objects directly into the immortal generation, pretenuring can not only save precious copying time at every garbage collection, but also increase the overall usable heap size. This is because in any type of GC that uses copying, including an Appel-style collector [3] which was used in the experiments, allocating objects in the mature space requires the system to keep a copy reserve of at least the same size as the mature space, essentially doubling the amount of space required for the mature space (see Chapter 2). Objects allocated in the immortal space, however, do not have this effect since they will never have to be moved.

Experimental results both in this thesis (see Chapter 4) and in Blackburn et al.'s original papers [26, 22] indicate that pretenuring schemes can considerably reduce garbage collection time by up to 80%. Recall however that pretenuring short-lived objects into the mature-space or the immortal space can decrease performance and diminish the effective space available in the heap. It is therefore essential for any pretenuring scheme to provide conservative advice.

Prior to this thesis, the research had investigated ways to produce the best possible pretenuring decisions by recording and analysing program traces [26, 22]. However, performing such analysis, could take days for each program and would require very large tracefiles (2 GB compressed files are not uncommon) to be recorded and analysed.

This thesis reviewed a substantially faster way of performing lifetime analysis using class-level static properties. This lifetime advice was successfully exploited to show important performance improvements when using the right kind of predictors. This novel methodology is summarised in the following section.

7.1.2 Methodology

As explained in Chapter 3, allocation sites are first classified as short-lived, long-lived or immortal, following Blackburn et al.'s methodology [26, 22] . This step is done by recording tracefiles and then analysing them. During this process, each allocation site is classified as allocating mostly short-lived, long-lived or immortal objects.

The predictors of both source and destination classes are then recorded at each allocation site, along with the expected lifetime. A data-mining analysis allows us to derive rules matching certain predictors with lifetimes at different levels of confidence. This information is stored in a knowledge bank. Unlike previous approaches, this analysis is performed once, and from that point onwards, it is possible to predict the lifetime of objects within programs never encountered before based on the predictors they exhibit.

Finally, before running a program, the source and destination predictors are checked at every allocation site within that program and generate lifetime advice for every object by matching their predictors against the knowledge bank of rules. This analysis is cheap and can be performed within seconds.

7.1.3 Software Metrics

Software metrics were developed in an attempt to produce a concise and quantitative analysis of certain aspects of a software system. They help to describe the quality and complexity of a software system in an impartial and objective way. The use of software metrics as predictors is analysed, in Chapter 5.

The information theoretic analysis (see Chapter 5.2) showed that metrics do not correlate very well with lifetimes, and that using combinations of metrics, especially combining source metrics with destination metrics, improves the correlation only slightly.

Various approaches were explored, including using only the top four or top six metric values in the analysis. Different methods of discretising software metrics

values into more usable data were also explored. Unfortunately, all the experimental results indicate that the *CK* software metrics are not good lifetime predictors.

The fact that CK metrics are not good predictors can be explained by a combination of several factors specific to software metrics. Firstly, as shown in Chapter 5, software metric ranges can be very wide (the RFC metric in the training set ranges from 1 to 1,370). Secondly, CK Metrics are general purpose metrics. They were created in order to produce a concise and quantitative analysis of certain aspects of a software system. They define and quantify interactions between classes. However, they offer no insight on the programmer's intent as they do not focus on class level properties, but rather on inter-class relationships.

7.1.4 Micro Patterns

Chapter 6 showed how micro-patterns, which allow the mechanical classification of Java classes, can be successfully used as lifetime predictors. Micro-patterns are defined as "a non-trivial, formal condition on the attributes, types, name and body of a class and its components, which is mechanically recognisable, purposeful, prevalent and simple" [62]. They are similar to design patterns, but closer to the implementation and describe properties of a single class.

It was shown that programmer's intentions can be captured with micro-patterns, applied to object allocation sites. By data mining a large corpus of Java programs, relationships between patterns exhibited at an allocation site and the lifetimes of the objects allocated by that site are found.

This analysis is effective at discovering short-lived and immortal objects, but predicts fewer sites that allocate long-lived objects. Experimental results show impressive performance gains in terms of GC time of between 6 and 77% in GC time, against a generational copying collector for several jvm98 programs. It was also shown that pause-times can be reduced by up to 546%.

With regards to the accuracy of the advice, advice generated at 75% confidence matched self-prediction advice 81% of the time on average.

Micro-patterns are good object lifetimes predictors because they are capable

of capturing programmer intentions. Furthermore, they are not subject to interpretation, which makes them easily exploitable. Finally, the data-mining classification process is much easier using micro-patterns than software metrics because micro-patterns are binary: a class exhibits a micro-patterns or it does not.

7.2 Discussion

This thesis demonstrated that it is possible for a fast static analysis to accurately predict the lifetime of objects. By knowing ahead of time the lifetime of an object, it is possible to make some pretenuring decisions and reduce the overall GC time, as well as the number and the length of pause times.

7.2.1 Predictors

One of the predictors, software metrics, did not produce good results. It was shown that software metrics ranges can be wide. Despite several different approaches, discretising effectively the data-set into meaningful values is hard. *CK* metrics are also general purpose metrics. They define interaction between classes and do not capture programmer intent.

Using micro-patterns as a predictor, on the other hand, proved very successful, with speedups in GC time by up to 77%. There are several reasons for micro-patterns giving better predictions than software metrics. First, micro-patterns are boolean values: a class exhibits a certain micro-pattern or it does not. Second, there are 30 different micro-patterns, providing a combination of $30 * 30 = 900$ micro-patterns if both source and destination are considered, which helps discriminate allocation sites. Finally, micro-patterns are capable of capturing the *intent* of the programmer by capturing common coding practices (such as controlled creation, interfaces, etc.) which software metrics are unable to do.

7.2.2 Limitations

The work presented in this thesis is novel and results obtained with our training and testing set are good. However, it is important to note that the technique developed in this thesis is not applicable to all types of programs. Below is a non-exhaustive list of cases where this technique is not recommended:

- *Long-running programs.* Long-running programs such as application servers may run for many months in a row without being restarted. If using the technique advocated in this thesis, certain objects will be allocated in the immortal space. However, if too many objects are allocated in the immortal region over the days, the program may run out of memory. To prevent this from happening, it is advisable to pretenure all objects predicted immortal into the mature space so that they can be reclaimed once the mature space becomes too large.

- *Critical applications.* Another type of applications where this technique is not advisable are critical applications that cannot tolerate failure. Examples of such applications can be found in the aeronautics industry, the space industry, the nuclear and energy industry, transports and many more. For these critical applications, as with long-running programs, it is recommended to allocate all objects predicted immortal in the mature space so that they can be reclaimed if the mature space becomes too large.

- *Programs with phases.* This technique may not apply well to programs that change behaviour throughout execution (phases). Objects predicted immortal may become dead when the program enters a new phase.

- *Programs with dramatically varying inputs.* This technique was successfully tested against a variety of difference programs loading a variety of different inputs. However, it is possible that certain programs capable of loading very different inputs may dramatically change behaviour based on the input

being loaded. For these programs, the technique developed in this thesis may not be applicable.

7.3 Future Work

This thesis demonstrated that class level properties can be exploited to predict object lifetimes. In particular, it was shown that micro-patterns are good lifetime predictors. In contrast, interaction between classes using software metrics does not give good results.

This section presents key areas of future work. Section 7.3.1 discusses finding lifetime-specific micro-patterns. Section 7.3.2 discusses how pointer analysis could potentially be used to predict object lifetimes. Section 7.3.3 discusses incorporating the decision making process inside the class-loader. Finally, Section 7.3.4 discusses the possibility of making more detailed predictions and using a specially designed garbage collector capable of taking advantage of this.

7.3.1 Exploring Lifetime-Specific Micro-Patterns

Gil and Maman [62] identified 30 micro-patterns (see Table 16). Micro-patterns were identified in a logical manner in order to capture common coding practices.

Each Java class has a set of characteristics, and the presence of a micro-pattern in a class means that the class exhibits several characteristics. Currently, Gil and Maman's catalog of micro-patterns capture only a small set of all possible combinations of characteristics.

With regards to predicting lifetimes, certain combinations of characteristics not part of Gil and Maman's micro-patterns catalogue could have a high correlation with the lifetime of objects. The discovery of new GC-specific micro-patterns could add great value to the existing system and reduce further the time spent in GC by the JVM.

The procedure leading to the discovery of GC-specific micro-patterns would be as follows:

1. List all different characteristics that a class can exhibit. Note that while a characteristic like "*the class has more than 1 private method*" is acceptable, "*the class has 3 private methods*" is not advisable as the number of characteristics would be infinite.

2. Calculate every combination of characteristics possible.

3. Evaluate each combination to see how frequently it is used within programs, and how well it correlates with object lifetimes.

By dramatically increasing the number of micro-patterns in the catalogue, the hope is to find more micro-patterns correlating highly with object lifetimes. This would help improve the quality of the advice, which would then lead to more important time savings.

The next section discusses the use of static analysis techniques to predict object lifetimes.

7.3.2 Pointer Analysis

Chapter 6 showed how micro-patterns can be exploited to predict object lifetimes. To discover micro-patterns within a program, Gil and Maman's tool analyses Java byte-code to identify the properties of each class. This task is performed statically offline. Experimental results indicate that micro-patterns are good lifetime predictors.

Since static analysis of class-level properties (micro-patterns) can predict object lifetimes with high accuracy, other static analysis techniques may be capable of predicting the lifetime of objects. A particular area of interest for future work is pointer analysis, a static code analysis technique that aims at discovering which pointers, or heap references, can point to which objects. Many publications on this topic exist, and much research remains to be done. Hind summarises the current state of the field in his paper [78].

Using pointer analysis, it may be possible to discover statically the lifetime of some objects. For example, if the pointer analysis can determine that an object

is used throughout the code until the end of the program, then this object is immortal. Likewise, if the analysis reveals that an object does not escape the method's scope, then it may be short-lived.

However, statically measuring the lifetime of objects that escape the scope of the method they were allocated in is difficult [78]. Therefore, the main goals of this approach are expected to lie in finding immortal and short-lived objects which do not escape the scope of the method they were allocated in.

The next section analyses the possibility of integrating the decision making engine inside the class-loader.

7.3.3 Decision Making Inside the Class-Loader

In the current setup, source predictors and destination predictors for each allocation site are discovered offline, before running the program. Source predictors and destination predictors are then matched against the knowledge bank, and an advice file is created, matching allocation sites with lifetime advice. Finally, the advice file is loaded into the JVM before the Java program starts.

In order to take advantage of lifetime predictions, the methodology currently imposes waiting for the above process to be completed before starting the program. While waiting a few minutes before starting the program is acceptable in the context of research, it is not acceptable for end-users, although it would only need to be done once.

In order to improve the usability of the system, the decision making engine could be incorporated inside the class loader, instead of having to generate predictions prior to running their program. Every time a class is loaded into the class-loader, the source predictors and destination predictors of each allocation site would need to be checked and matched against a local knowledge bank. For the end-user, this scheme would be transparent, and the user would not have to wait minutes for the advice to be generated before starting their program. The cost of the analysis would be distributed throughout the program's execution (whenever a class is loaded into memory), as opposed to be an upfront cost as it

currently is.

However, for such implementation to be successful, it is extremely important to ensure that the overhead on the system is limited, as the extra overhead could easily outweigh the benefits of pretenuring. King shows how this analysis could be run in the background without the need for thread synchronisation at GC time [98]. The advice generated by this analysis could then be fed into the optimising compiler to perform pretenuring decisions.

7.3.4 A More Suitable Garbage Collector

As explained in Chapter 3, the prototype was implemented inside a standard Appel-style collector [3]. These are classified as short-lived, long-lived or immortal following Blackburn et al.'s classifications. This garbage collector is well suited to take advantage of the predictions since long-lived objects can be allocated directly into the mature space, and immortal objects can be allocated directly into the immortal space.

Jones and Ryder propose a lifetime aware collector which is capable of taking advantage of more precise object lifetimes predictions [92]. Using appropriate predictors such as micro-patterns, it may be possible to predict the lifetime of objects more accurately. By using a different lifetime classification than used in this thesis, it may be possible to take advantage of a lifetime aware garbage collector .

7.3.5 A Larger Training Set

This work was carried out using a limited training set and test set. While this is enough to prove the viability of the approach, using a larger training set may lead to significant improvements in performance. A larger training set taken from real-life applications may allow the discovery of new rules capable of predicting object lifetimes with high accuracy.

In general, the larger the training set is, the more generic the knowledge bank

would become. This would lead to an increased probability of accurately predicting the lifetime of objects in programs never encountered before. This would also lead to the discovery of more rules, increasing the probability of being able to predict the lifetime of particular objects.

7.3.6 Final Words

This thesis discussed a novel approach to object lifetime prediction for pretenuring by combining data-mining techniques with static analysis. In particular, this thesis demonstrated some important speedups in GC time using micro-patterns as predictors.

Nevertheless, a lot of work remains to be done, and several approaches can be taken to improve the system, both in terms of usability and prediction accuracy. Having shown that static properties can be good indicators of object lifetimes, a few questions remain:

- "What is the most accurate predictor in the Java language?"
- "Which predictor offers the best accuracy versus cost ratio?"

Bibliography

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño virtual machine. In *IBM System Journal, Vol 29, No 1.* IBM, 2000.

[2] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

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

[4] Andrew W. Appel. Compilers and runtime systems for languages with garbage collection. In *Proceedings of SIGPLAN'92 Conference on Programming Languages Design and Implementation*, volume 27 of *ACM SIGPLAN Notices*, San Francisco, CA, June 1992. ACM Press.

[5] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. *ACM SIGPLAN Notices*, 26(4):96–107, 1991. Also in SIGARCH Computer Architecture News 19 (2) and SIGOPS Operating Systems Review 25.

[6] Alain Azagury, Elliot K. Kolodner, Erez Petrank, and Zvi Yehudai. Combining card marking with remembered sets: How to save scanning time. In Jones [91], pages 10–19.

[7] Hezi Azatchi and Erez Petrank. Integrating generations with advanced reference counting garbage collectors. In *Proceedings of the Compiler Construction: 12th International Conference on Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 185–199, Warsaw, Poland, May 2003. Springer-Verlag Heidelberg.

[8] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.

[9] David F. Bacon, Perry Cheng, and V.T. Rajan. A unified theory of garbage collection. In OOPSLA [118].

[10] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In Jørgen Lindskov Knudsen, editor, *Proceedings of 15th European Conference on Object-Oriented Programming, ECOOP 2001*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, June 2001. Springer-Verlag.

[11] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[12] Henry G. Baker. 'Infant mortality' and generational garbage collection. *ACM SIGPLAN Notices*, 28(4), April 1993.

[13] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of Roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[14] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In PLDI [125], pages 187–196.

[15] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[16] E. C. Berkeley and Daniel G. Bobrow, editors. *The Programming Language LISP: Its Operation and Applications*. Information International, Inc., Cambridge, MA, fourth edition, 1974.

[17] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Conference Record of the Twenty-third Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, 1996.

[18] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, October 2007.

[19] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[20] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, October 2006. ACM Press.

[21] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE*

2004, 26th International Conference on Software Engineering, Edinburgh, May 2004.

[22] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, 29(1):1–57, 2007.

[23] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In PLDI [126], pages 153–164.

[24] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In OOPSLA [117].

[25] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, 2008.

[26] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinley, and J. Eliot B. Moss. Pretenuring for Java. In OOPSLA [115], pages 342–352.

[27] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[28] Hans-Juergen Boehm. Reducing garbage collector cache misses. In Hosking [83].

[29] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[30] G. Bollella and J. Gosling. The real-time specification for Java. *Computer*, 33(6):47–54, Jun 2000.

[31] boost.org. *Boost C++ Libraries*. Boost.org, 2009.

[32] David R. Brownbridge. *Recursive Structures in Computer Systems*. PhD thesis, University of Newcastle upon Tyne, September 1984.

[33] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Vivek Sarkar Michael Hind, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141. ACM Press, 1999.

[34] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Barcelona Spain, 2001.

[35] D. C. Cann and Rod R. Oldehoeft. Reference count and copy elimination for parallel applicative computing. Technical Report CS–88–129, Department of Computer Science, Colorado State University, Fort Collins, CO, 1988.

[36] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In Norman Meyrowitz, editor, *OOPSLA'86 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 21(11) of *ACM SIGPLAN Notices*, pages 119–130. ACM Press, October 1986.

[37] *Proceedings of the 14th International Conference on Compiler Construction*, Edinburgh, April 2005. Springer-Verlag.

[38] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[39] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[40] Chen-Yong Cher, Antony L. Hosking, and T. N. Vijaykumar. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. *SIGOPS Oper. Syst. Rev.*, 38(5):199–210, 2004.

[41] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In Diwan [53].

[42] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[43] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. Technical report, National University of Singapore, November 2003.

[44] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 243–254, Washington, DC, June 2004. ACM Press.

[45] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in Lisp. *Communications of the ACM*, 20(2):78–86, February 1977.

[46] David Cohn and Satinder Singh. Predicting lifetimes in dynamically allocated memory. In M. Mozer et al., editors, *Advances in Neural Information Processing Systems 9*, 1997.

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

[48] David Detlefs, editor. *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[49] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele. Lock-free reference counting. *Distributed Computing*, 15:255–271, 2002.

[50] John DeTreville. Experience with garbage collection for Modula-2+ in the Topaz environment. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[51] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 157 – 168, San Francisco, CA, January 1990. ACM Press.

[52] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[53] Amer Diwan, editor. *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

[54] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Vancouver, June 2000. ACM Press.

[55] Robert R. Fenichel and Jerome C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[56] François Fleuret. Fast binary feature selection with conditional mutual information. *J. Mach. Learn. Res.*, 5:1531–1555, 2004.

[57] John K. Foderaro and Richard J. Fateman. Characterization of VAX Macsyma. In *1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 14–19, Berkeley, CA, 1981. ACM Press.

[58] William J. Frawley, Gregory Piatetsky-Shapiro, and Christopher J. Matheus. Knowledge discovery in databases: An overview. In *The American Association for Artificial Intelligence*, pages 57–70, 1992.

[59] Yoav Freund. Boosting a weak learning algorithm by majority. In *Information And Computation*. ACM, 2008.

[60] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[61] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. Effective prefetch for mark-sweep garbage collection. In *The 2007 International Symposium on Memory Management*. ACM Press, October 2007.

[62] Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In *Object-Oriented Programming, Systems, Language and Applications (OOPSLA'05)*, pages 97–116, San Diego, CA, 2005. ACM Press.

[63] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[64] Udo Grimmer. Clementine: Data mining software. In Hans-Joachim Mucha and Hans-Hermann Bock, editors, *Classification and Multivariate Graphics: Models, Software and Applications*, number 10, pages 25–31. Weierstrass-Institut für Angewandte Analysis und Stochastik, Berlin, 1996.

[65] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In PLDI [125], pages 177–186.

[66] Samuel Guyer and Kathryn McKinley. Finding your cronies: Static analysis for dynamic object colocation. In OOPSLA [118].

[67] N. Hallenberg. Combining garbage collection and region inference in the ML Kit. Master's thesis, Department of Computer Science (DIKU), University of Copenhagen, June 1999.

[68] Niels Hallenberg. A region profiler for a Standard ML compiler based on region inference. Student Project 96–5–7, Department of Computer Science (DIKU), University of Copenhagen, June 1996.

[69] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In PLDI [126], pages 141–152.

[70] D.J. Hand. Data mining: Statistics and more? In *The American Statistician*, number 52, 1998.

[71] Lars Thomas Hansen. *Older-first Garbage Collection in Practice.* PhD thesis, North-eastern University, November 2000.

[72] Lars Thomas Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP02)*, volume 37(9) of *ACM SIGPLAN Notices*, pages 247–258, Pittsburgh, PA, 2002. ACM Press.

[73] Timothy Harris. Dynamic adaptive pre-tenuring. In Hosking [83].

[74] Barry Hayes. Using key object opportunism to collect old objects. In Andreas Paepcke, editor, *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[75] Matthew Hertz, Steve M. Blackburn, K. S. McKinley, J. Eliot B. Moss, and Darko Stefanovic. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, Marina Del Rey, CA, June 2002.

[76] Michael Hicks, Luke Hornof, Jonathan T. Moore, and Scott Nettles. A study of Large Object Spaces. In Jones [91], pages 138–145.

[77] M. Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[78] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, June 2001.

[79] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In OOPSLA [117].

[80] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In Detlefs [48], pages 36–49.

[81] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[82] Antony L. Hosking and Richard L. Hudson. Remembered sets can also play cards. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[83] Tony Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[84] Wei Huang, W. Srisa-an, and J. M. Chang. Adaptive pretenuring schemes for generational garbage collection. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-04)*, pages 133–140, Austin, TX, March 2004.

[85] Xianlong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Z. Wang, and Perry Cheng. The garbage collection advantage: Improving program locality. In OOPSLA [118].

[86] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Department of Computer and Information Science, September 1991.

[87] R. John M. Hughes. A semi-incremental garbage collection algorithm. *Software Practice and Experience*, 12(11):1081–1084, November 1982.

[88] H. Inoue, Darko Stefanović, and S. Forrest. Object lifetime prediction in Java. Technical Report TR–CS–2003–28, University of New Mexico, May 2003.

[89] Hajime Inoue, Darko Stefanovic, and Stephanie Forrest. On the prediction of Java object lifetimes. *IEEE Transactions on Computers*, 55(7):880–892, 2006.

[90] Douglas Johnson. The case for a read barrier. *ACM SIGPLAN Notices*, 26(4):279–287, 1991.

[91] Richard Jones, editor. *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, Vancouver, October 1998. ACM Press.

[92] Richard Jones and Chris Ryder. Garbage collection should be lifetime aware. In Olivier Zendra, editor, *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, page 8, Nantes, France, July 2006.

[93] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[94] Richard E Jones and Chris Ryder. A study of java object demographics. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 121–130, New York, NY, USA, 2008. ACM.

[95] Sebastien Marion Richard Jones and Chris Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In Mooly Sagiv, editor, *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, pages 67–78, Montréal, Canada, October 2007. ACM Press.

[96] Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In Diwan [53].

[97] C. F. Kemerer. Reliability of function points measurement: A field experiment. In *Commun*, pages 85–97, 1993.

[98] Andy C. King. Removing GC synchronisation. In OOPSLA [116], pages 112–113 (Companion).

[99] Donald E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms, chapter 2. Addison-Wesley, second edition, 1973.

[100] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. pages 1137–1143. Morgan Kaufmann, 1995.

[101] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:77–86, 1951.

[102] T. Kurokawa. A new fast and safe marking algorithm. *Software Practice and Experience*, 11:671–682, 1981.

[103] Yossi Levanoni and Erez Petrank. A scalable reference counting garbage collector. Technical Report CS–0967, Technion — Israel Institute of Technology, Haifa, Israel, November 1999.

[104] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In OOPSLA [115].

[105] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. *ACM Transactions on Programming Languages and Systems*, 28(1), January 2006.

[106] Henry Lieberman and Carl E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[107] Rafael D. Lins. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44(4):215–220, 1992. Also Computing Laboratory Technical Report 75, University of Kent, July 1990.

[108] Rafael D. Lins. An efficient algorithm for cyclic reference counting. *Information Processing Letters*, 83:145–150, 2002.

[109] A. D. Martinez, R. Wachenchauzer, and Rafael D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34:31–35, 1990.

[110] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[111] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[112] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3:184–195, 1960.

[113] SUN Microsystems. The Java HotSpot Virtual Machine, 1999. Technical White Paper.

[114] K.Nygaard O.Dahl, B.Myhrhaug. The Simula67 Base Common Base Language. Technical report, Norwegien Computing Center, 1970.

[115] *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[116] *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Seattle, WA, November 2002. ACM Press.

[117] *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[118] *OOPSLA'04 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Vancouver, October 2004. ACM Press.

[119] Patrik Paetau. *On the Benefits and Problems of the Object-Oriented Paradigm including a Finnish Study.* PhD thesis, Swedish School of Economics and Business Administration, October 2005.

[120] Harel Paz and Erez Petrank. Using prefetching to improve reference-counting garbage collectors. In *Proceedings of the 16th International Conference on Compiler Construction (CC'07)*, March 2007.

[121] Harel Paz, Erez Petrank, David F. Bacon, V.T. Rajan, and Elliot K. Kolodner. An efficient on-the-fly cycle collection. In CC [37].

[122] Harel Paz, Erez Petrank, David F. Bacon, V.T. Rajan, and Elliot K. Kolodner. An efficient on-the-fly cycle collection. *ACM Transactions on Programming Languages and Systems*, 29(4):1–43, 2007. Article 20.

[123] Harel Paz, Erez Petrank, and Stephen M. Blackburn. Age-oriented garbage collection. In CC [37].

[124] E. J. H. Pepels, M. C. J. D. van Eekelen, and M. J. Plasmeijer. A cyclic reference counting algorithm and its proof. Technical Report 88–10, Computing Science Department, University of Nijmegen, 1988.

[125] *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, NM, June 1993. ACM Press.

[126] *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Berlin, June 2002. ACM Press.

[127] Tony Printezis and Alex Garthwaite. Visualising the Train garbage collector. In Detlefs [48], pages 100–105.

[128] J. Ross Quinlan. Data mining tools See5 and C5.0.

[129] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[130] John H. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ, December 1993.

[131] Niklas Röjemo and Colin Runciman. Lag, drag, void, and use: heap profiling and space-efficient compilation revisited. In *Proceedings of First International Conference on Functional Programming*, pages 34–41, Philadelphia, PA, May 1996. ACM Press.

[132] Paul Rovner, Roy Levin, and John Wick. On extending Modula-2 for building large, integrated systems. Technical Report 3, DEC Systems Research Center, Palo Alto, CA, Palo Alto, CA, 1985.

[133] Christina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 285–293. ACM Press, January 1988.

[134] Colin Runciman and Niklas Röjemo. Lag, drag and post-mortem heap profiling. In *Implementation of Functional Languages Workshop*, Bøastad, Sweden, September 1995.

[135] Narendran Sachindran, J. Eliot B. Moss, and Emery D. Berger. MC^2: High-performance garbage collection for memory-constrained environments. In OOPSLA [118].

[136] Jon D. Salkild. Implementation and analysis of two reference counting algorithms. Master's thesis, University College, London, 1987.

[137] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.

[138] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, 1998.

[139] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of Java applications at allocation and garbage collection times. In OOPSLA [116].

[140] David Siegwart and Martin Hirzel. Improving locality with parallel hierarchical copying gc. In J. Eliot B. Moss, editor, *ISMM'06 Proceedings of*

the Fourth International Symposium on Memory Management, pages 52–63, Ottawa, Canada, June 2006. ACM Press.

[141] Jeremy Singer, Gavin Brown, Mikel Lujan, and Ian Watson. Towards intelligent analysis techniques for object pretenuring. In *Principles and Practice of Programming in Java*, Lisbon, September 2007. ACM Press.

[142] Jeremy Singer, Sebastien Marion, Gavin Brown, Richard Jones, Mikel Lújan, Chris Ryder, and Ian Watson. An information theoretic evaluation of software metrics for object lifetime prediction. In *2nd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART'08)*, page 15, Goteborg, Sweden, January 2008.

[143] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[144] Sunil Soman and Chandra Krintz. Application-specific garbage collection. *Journal of Systems and Software*, 80(7):1037 – 1056, 2007. Dynamic Resource Management in Distributed Real-Time Systems.

[145] Diomidis Spinellis. CKJM—Chidamber and Kemerer Java metrics, 2005. `http://www.spinellis.gr/sw/ckjm/`.

[146] IBM Staff. The jikes research virtual machine (rvm), 2000.

[147] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[148] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[149] Darko Stefanovic, Matthew Hertz, Stephen Blackburn, Kathryn McKinley, and J. Eliot Moss. Older-first garbage collection in practice: Evaluation in

a Java virtual machine. In *ACM SIGPLAN Workshop on Memory System Performance (MSP 2002)*, Berlin, June 2002.

[150] Bjarne Stroustrup. A history of C++: 1979-1991. *SIGPLAN Notices*, 28:271–297, 1993.

[151] G. Succi, W. Pedrycz, S. Djokic, P. Zuliani, and B. Russo. An empirical exploration of the distributions of the Chidamber and Kemerer object-oriented metrics suite. *Empirical Software Engineering*, 10(1):81–104, 2005.

[152] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison Wesley, 2005.

[153] Simon J. Thompson and Rafael D. Lins. Cyclic reference counting: A correction to Brownbridge's algorithm. Unpublished notes, 1988.

[154] Mads Tofte. A brief introduction to Regions. In *ISMM98*, pages 186–195.

[155] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998.

[156] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3), September 2004.

[157] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with Regions in the ML Kit, version 4. Technical report, IT University of Copenhagen, October 2001.

[158] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with Regions in the ML Kit. Technical Report DIKU-TR-97/12, Department of Computer Science (DIKU), University of Copenhagen, April 1997.

[159] Mads Tofte and Niels Hallenberg. Region-based memory management in perspective. In *Proceedings of the First workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*, London, January 2001. Invited talk.

[160] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report Computer Science 93/15, University of Copenhagen, July 1993.

[161] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 188–201, Portland, OR, January 1994. ACM Press.

[162] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, February 1997.

[163] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[164] David M. Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, 1988.

[165] David M. Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[166] Daniel C. Wang and Andrew W. Appel. Safe garbage collection = regions + intensional type analysis. Technical report, Princeton, July 1999.

[167] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, 1983.

[168] J. Weizenbaum. Recovery of reentrant list structures in SLIP. *Communications of the ACM*, 12(7):370–372, July 1969.

[169] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[170] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM Press.

[171] Benjamin Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.

[172] Benjamin G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms.* PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.

Appendix A

Information Theoretic Analysis Of CK Metrics

This Appendix presents the full details of the paper "An Information Theoretic Evaluation of Software Metrics for Object Lifetime Prediction" [142].

The lead author of this paper is Jeremy Singer from the University of Manchester with whom we were collaborating on this research. Singer also introduced the information theoretic approach.

The author of this thesis provided the idea of using data-mining, provided the scripts necessary to parse the data-mining rules and match them with predictors to generate lifetime predictions. The author of this thesis also performed the runtime measurements using the experimental JVM.

A.1 Data-mining algorithm used

Because this research was not carried out at the same university and because C5.0 is a commercial product, this part of the research was done using the data-mining algorithm C4.5 and not C5.0 that we use for the bulk of this thesis.

C5.0 is the evolution of C4.5 and improves on a few points. It performs faster than the previous version and uses less memory by using smaller decision trees. Also, it introduced support for boosting (see Chapter 2.5), weighting (see Chapter

2.5) and *winnowing* (a technique which helps reduce noise in the data). When there are numerous attributes, winnowing operates by pre-selecting a subset of attributes with valuable predictive information that will be used to construct the rule-set.

A.2 Calculation of Information Theory Measurements

The fundamental information theoretic measure is *entropy*, which quantifies the information content in a given source of data: the more 'randomness' or unpredictability in the data source, the higher the entropy value. Consider a device producing symbols according to a random variable X, defined over a finite alphabet of possible symbols S_X. If we assume each successive symbol $s_i \in S_X$ is independent of the previous ones, the *unconditional entropy* is defined as,

$$H(X) = -\sum_{i=1}^{|S_X|} p(i) \log(p(i)) \tag{6}$$

where $p(i)$ is the probability of the ith symbol being produced. Note that all logarithms are base 2. Inpractical terms, $p(i)$ can be calculated with frequency counts, i.e.:

$$p(i) = \frac{\text{number of occurrences of symbol } s_i}{\text{total number of symbols seen}} \tag{7}$$

We assume the produced symbols to be a common CK metric measurement on a sequence of dynamic object allocations. This gives us a stream of metric values, for which we can calculate an entropy measure.

The *conditional entropy* measures the dependence between two different symbol streams. In our case, we could take two measurements on each element in a sequence of dynamic object allocations. For instance we could measure a CK metric for each newly created object and its lifetime. These are two different random variables X and Y respectively with two different alphabets S_X and S_Y but there

may be some dependence between them, which we can quantify by conditional entropy.

$$H(Y|X) = -\sum_{i=1}^{|S_X|} p(i) \sum_{j=1}^{|S_Y|} p(j|i) \log(p(j|i)) \tag{8}$$

This is the *first order conditional entropy.* The required probabilities can again be computed from frequency counts:

$$p(j|i) = \frac{\text{number of times } s_j \text{ occurs with } s_i}{\text{number of occurrences of } s_i} \tag{9}$$

First order conditional entropy has a minimum value of zero and a maximum value of $\log(|S_Y|)$. In the example above, it measures the uncertainty we have in the lifetime of an object given the value of a certain CK metric. If lifetime values are produced uniformly at random over the alphabet S_Y, then eq.(8) will converge to $\log(|S_Y|)$.

The *mutual information* between X and Y is a measure of the agreement, or correlation, between them. The mutual information is,

$$I(X;Y) = H(Y) - H(Y|X) \tag{10}$$

This is easily computed from the entropy measurements we have already described above. This measurement is symmetric, i.e. $I(X;Y) = I(Y;X)$, and quantifies the reduction in our uncertainty of Y when the value of X is revealed. Unlike Pearson's R correlation coefficient, which only detects *linear* correlations between random variables, mutual information can detect arbitrary *nonlinear* relationships.

$I(X;Y)$ can be normalized to a value between 0 and 1 by dividing it by $\min(H(X), H(Y))$. The maximum value of normalized mutual information (NMI) indicates that there is perfect correlation between the two variables. Given the value of one variable, it is always theoretically possible to construct a predictor that will predict the value of the other variable with 100% accuracy. A low value of NMI indicates that there is little information, and therefore little opportunity for accurate prediction of Y given X. A zero value of NMI indicates that the two

variables are entirely uncorrelated, so knowing the value of one variable does not avail for making predictions about the other variable's value.

A.3 Correlation of Individual features

Our first analysis assesses the utility of single CK metrics as features for predicting object lifetimes. The NMI scores are shown in Table 13. The rows are sorted according to NMI values. Each row reports the NMI of a single metric with the object lifetime. Most of the figures are disappointingly low. For instance, knowing the NOC metric (number of child classes) for source and destination classes is almost useless for predicting lifetimes. Other metrics show limited potential: six metrics have NMI scores above 0.25.

The top two metrics in Table 13 are CK metrics for source objects. We might assume that it is more important to know about the source object than the destination object, or at least, it is important to know something about the source object as well as the destination object. This is an important insight: until now, most type-based object lifetime studies only consider characteristics of the destination object (the allocatee) rather than the source (the allocator).

A.4 Cross-Correlation

Given the low NMI scores of single metrics with lifetime, one might ask if a *combination* of metrics is required to obtain accurate predictions. When selecting a set of features to supply as input to a predictor, the following heuristic should be followed: "Selected features should have low correlations with each other (cross-correlation) and high correlation with the output." So in our case, we need to identify CK metrics that do not correlate highly with other metrics, but have relatively high correlation with object lifetime.

Table 14 shows the cross-correlation values of metrics with other metrics. A metric's cross-correlation with itself is always 1, hence the unit diagonal in the

matrix. Since NMI is a symmetric score, then the matrix is symmetric about its diagonal, so the table only reports the upper triangle of values.

This table shows that source metrics correlate highly with other source metrics, and destination metrics correlate highly with other destination metrics. However, there is a low correlation between source metrics and destination metrics in general. Therefore a good selection of metrics for predictor inputs would be a *combination* of source and destination metrics.

A.5 Correlation of Pairs of Features

Table 22 gives the NMI correlation scores of pairs of features with object lifetime. We only consider single features that have correlation above 0.25 with NMI from Table 13. It is important to note that this table conveys different information from Table 14. Now the row X and column Y indicate metrics X and Y whose values are paired. The table cell value for (X,Y) is the NMI of this pair of metrics with lifetime. So the diagonal values of pairs (X,X) are the same as the values for individual metrics in Table 13. The other values show the correlation of pairs of features with lifetime. Note that using one source metric and one destination metric (top right hand corner of table) gives significantly better NMI scores than using two source metrics (top left hand corner of table) or two destination metrics (bottom right corner of table). This is empirical confirmation of the notion we outlined in the previous section—that a mixture of source and destination metrics are better than all source or all destination metrics alone.

A.6 Conditional Mutual Information Maximisation

Table 13 reveals that single features all have low NMI scores. We require a *combination* of features to obtain reliable predictions. However one problem with selecting features based only on NMI is that this selection process does not take

	sRFC	sLCOM	dWMC	dRFC	dLCOM	dNPM
sRFC	0.342	0.531	0.660	0.722	0.666	0.642
sLCOM		0.370	0.656	0.699	0.661	0.637
dWMC			0.257	0.412	0.389	0.350
dRFC				0.314	0.443	0.426
dLCOM					0.324	0.401
dNPM						0.261

Table 22: Correlation of pairs of features with lifetime.

account of cross-correlation between features. It is best to select features that have high individual correlation with the class to predict and have low cross-correlation with each other. Fleuret [56] presents an attractive algorithm to do automatic feature selection based on mutual information that considers cross-correlation. His technique is known as *conditional mutual information maximization* (CMIM). The approach iteratively picks features that maximize their mutual information with the class to predict, conditioned on features already picked. This CMIM criterion does not select a feature *similar* to ones already picked, even if it is individually informative, since such a similar feature does not carry *additional* information about the class to predict.

Conditional mutual information is calculated as:

$$I(U;V|W) = H(U|W) - H(U|W,V) \tag{11}$$

This value is an estimate of the quantity of information shared between U and V when W is known. If V and W carry the same information about U, then the two terms on the right are equal and the conditional mutual information is zero, even if both V and W are individually informative. Conversely if V contains information about U which is not present in W, then the difference is large and the conditional mutual information is high.

The CMIM algorithm operates as follows. It aims to pick k features from a

metric	*CMIM score*
source LCOM	0.407
dest RFC	0.345
source RFC	0.177
source CBO	0.144
dest LCOM	0.142
source NPM	0.113
source WMC	0.104
source Ca	0.086
dest NPM	0.085
dest WMC	0.072
dest Ca	0.061
source DIT	0.041
dest DIT	0.038
dest CBO	0.033
source NOC	0.025
dest NOC	0.007

Table 23: CMIM-based ranking of features for prediction of object lifetime.

total of n, in order of relevance with the most relevant feature first. Incidentally, if we use CMIM to select n features from n, then we get a relevance-ordered ranking of our entire feature set which takes into account cross-correlations in a way that our simple ranking based on NMI scores in Table 13 did not.

The algorithm maintains a score vector, with one element for each feature. Initially, the score vector is set up so $score[i]$ contains the *unnormalized* mutual information score for the ith feature (with the lifetime). At each iteration, the feature with the highest score value is taken as the next selected feature. Then the score vector is recomputed, with each element $score[i]$ set to the minimum value of $(score[i], I(\text{lifetime}; i\text{th feature}|\text{last selected feature})$. This ensures that $score[i]$ is low if at least one of the features already picked is similar to the ith feature.

Fleuret [56] gives a full explanation of the algorithm, including various optimization techniques using lazy evaluation and boolean bit-vectors. We implement his CMIM algorithm and use it to rank the CK metric features in order of relevance for object lifetime prediction. Table 23 gives the results of this CMIM analysis. Note that in the table, each score is the highest value in score vector at that particular iteration, for a feature that has not been selected by previous iterations.

A.7 Prototype Prediction Schemes

In order to evaluate the feature selection and ranking decisions above, we generate several predictors using the C4.5 tree learner algorithm. We report the accuracy of these decision tree predictors, but as yet we have not used the predictions to optimize generational GC, so we are unable to give real benchmark speedups at this stage.

Recall that the total object lifetime database contains around 40,000 entries. Each entry records source and destination CK metrics for a single scalar allocation site, together with the most likely lifetime for objects allocated at that site. The data is randomly split 50:50 into training/test sets. This is repeated for five trials. To use just five trials may not seem enough for statistical significance. However the results in all our experiments have such low variance that more than five trials is not necessary.

We investigate how the (mean) accuracy of the generated decision tree varies as different numbers of features are added. The features are selected according to their ranking from the CMIM algorithm, as shown earlier in Table 23. Figure 48 presents the results, including error bars to indicate plus/minus one standard deviation. It is evident that a tree built using all 16 features has an accuracy of approximately 78%. This is significantly better than the performance of the Weka baseline ZeroR predictor, which has an accuracy score of 44.8%. The ZeroR predictor always selects the most frequently occurring outcome, assuming all allocation sites are equally likely.

Note that when we select just the top three features indicated by CMIM, the generated predictor achieves 77.6% accuracy, for a significantly reduced decision complexity. After pruning, the three-feature trees had on average just 10 nodes, while the 16 feature trees had on average 40 nodes. It is clear to see that three features provide similar accuracy to 16 features, at a much lower complexity cost. This is clear justification for the application of feature selection techniques outlined earlier.

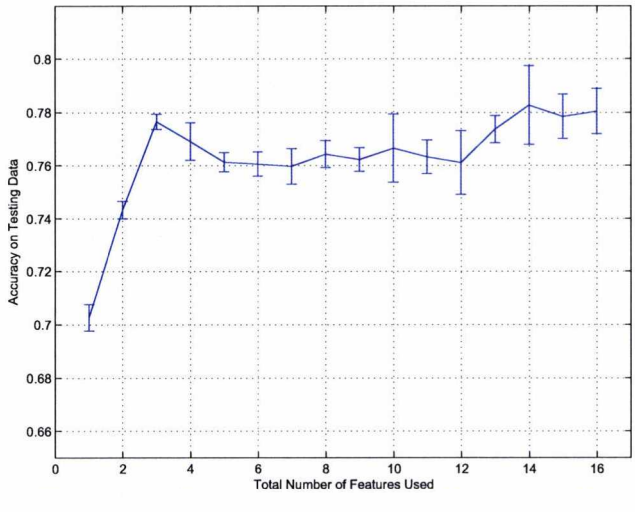


Figure 48: Graph showing how C4.5 predictor accuracy changes with number of metric features for lifetime prediction

As a further extreme example, we consider only the top *two* features identified by CMIM. This feature reduction, combined with aggressive pruning, enables us to visualize the decision boundaries of a small C4.5 decision tree. Figure 49 illustrates this map. It is a graphical presentation of a simple set of rules, that achieves 75% accuracy on the test data. The primary advantage of this visualization is that it can also support *interpretation* of the rules. We present three 'intuitive' instantiations of the rules below:

1. The point marked + in the map corresponds to an allocation site in the Dacapo `pmd` benchmark. The source class is `pmd.ast.JavaParser`. The destination class is the `LookaheadSuccess` inner class of the source. The map shows that this allocation is predicted to be *short-lived.* This seems likely, given domain-specific knowledge that look-ahead events are frequent in the parsing process, and the specific inner class contains no long-term state.

2. The * point corresponds to an allocation site in DaCapo `bloat`. The source class is `bloat.tree.PrintVisitor`. The destination class is `StringBuffer`.

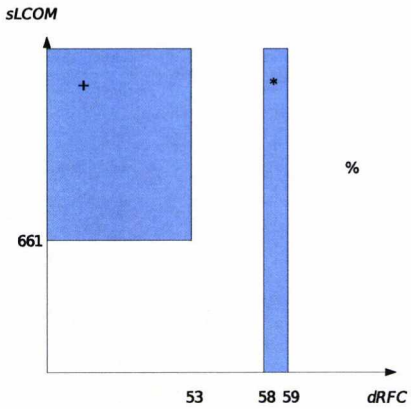


Figure 49: Two-dimensional map showing how allocation sites with various metric values map onto different lifetimes.
Shaded areas represent short-lived allocations. Unshaded areas represent immortal allocations.

This allocation is predicted to be *short-lived.* This seems appealing, since `StringBuffer` objects are generally ephemeral and the `PrintVisitor` class presumably makes a traversal over the entire tree, creating and emitting a textual representation of the tree nodes. We speculate that this narrow band of short-lived objects between 58 and 59 on the dRFC axis is almost entirely due to `StringBuffer` objects.

3. The `%` point corresponds to any allocation site whose destination class is `String`. All such allocations are predicted to be *long-lived.* Again, this appeals to intuition since `String` objects are immutable and often contain long-term information.

When using rules (as explained in Chapter 3) a single rule is equivalent to a single path from the root node to a leaf node in the decision tree. Some paths match many cases in the dataset, whereas others only have a single match. Some paths have 100% successful prediction rate, whereas other paths have lower accuracy. Selecting a subset of rules from the decision tree enables us to eliminate unpopular or inaccurate decisions. In addition, single rules are easier to interpret than a complete decision tree.

A.8 Explanation of Analysis

Feature selection is a fundamental topic in Machine Learning. Too many features can lead to *overfitting* of a learning model, and hence poor performance when the learning system is deployed in the field. It is important to distinguish here between feature *selection*, and feature *extraction*. The former is the focus of this section. The latter encompasses techniques such *Principal Components Analysis* (PCA) and *Bayesian Automatic Relevance Determination* (ARD) [18].

Extraction techniques measure *functions* of features, which are linear for PCA and nonlinear for ARD. The typical result is a 'black-box' mathematical function of the original data. The *meaning* of the original features is lost. In contrast, feature selection techniques such as those based on conditional mutual information maximization allow us to *retain original meaning* and provide human-readable explanations of *how* a feature is useful in combination with others.

For instance, we can attempt to interpret the feature ranking provided by CMIM in Table 23. This shows us that LCOM and RFC are important metrics, and that source metrics are generally more important than destination metrics. Note how there is only one out of eight CK metrics for which the destination value is ranked above the source value in the table.

The LCOM and RFC metrics measure the complexity of a class, in terms of methods. Recall that LCOM measures how various sets of methods in a class access disjoint field sets in the class, and RFC measures how many methods are called directly by the methods of a class. These two values provide a precise way to characterize a class. Perhaps this kind of precise metric 'fingerprint' what is needed to get accurate object lifetime predictions.