UNIVERSITY OF CALIFORNIA
Santa Barbara

Memory Management for Multi-Application Managed Runtime Environments

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sunil Soman

Committee in Charge:

Professor Chandra Krintz, Chair

Professor Tevfik Bultan

Professor Giovanni Vigna

June 2008

The Dissertation of
Sunil Soman is approved:

Professor Tevfik Bultan

Professor Giovanni Vigna

Professor Chandra Krintz, Committee Chairperson

May 2008

Memory Management for Multi-Application Managed Runtime Environments

Copyright © 2008

by

Sunil Soman

Acknowledgements

Graduate school is a process with hurdles, milestones, disappointments and elation. I would like to thank everyone who has made this process less painful and provided support, friendship, guidance and understanding over the years.

First, I would like to thank my advisor Chandra Krintz for always believing in me, supporting me, and always being available when I needed help. One could not wish for a better philosopher, guide and a friend. I thank Rich Wolski for being a mentor, philosopher and a friend. Chandra and Rich, thanks for everything.

I am grateful for exceptional friends like Chris, Dan, Dmitrii and Graziano. I would like to thank my mentor and collaborator Laurent Daynès at Sun Microsystems for hosting me while I was an intern and providing hours of technical discussion, support and guidance on a significant portion of work that is part of this dissertation.

I would also like to thank my colleagues in the RACE and Mayhem labs. They made the process more bearable and often enjoyable.

Finally, I would like to thank the staff, faculty and fellow graduate students at the Computer Science department at UC Santa Barbara who have made this journey memorable and fruitful, despite the challenges.

Curriculum Vitæ

Sunil Soman

Education

2008	Doctor of Philosophy in Computer Science, University of California, Santa Barbara.
2007	Master of Science in Computer Science, University of California, Santa Barbara.
2001	Bachelor of Engineering in Computer Engineering, University of Pune, India.

Experience

2007 – 2008	Graduate Intern, Sun Microsystems Labs.
2005 – 2005	Graduate Intern, Sun Microsystems Labs.
2002 – 2007	Research Assistant, University of California, Santa Barbara.
2001 – 2002	Teaching Assistant, University of California, Santa Barbara.

Publications

Sunil Soman, Chandra Krintz, Laurent Daynès, MTM2: Scalable Memory Management for Multi-Tasking Managed Runtime Environments, The 22nd European Conference On Object-Oriented Programming (ECOOP), July 2008

Chris Grzegorczyk, Sunil Soman, Chandra Krintz, and Rich Wolski, Isla Vista Heap Sizing: Using Feedback to Avoid Paging, The International Symposium on Code Generation and Optimization (CGO), March 2007

Sunil Soman and Chandra Krintz, Application-Specific Garbage Collection, The Elsevier Journal of Systems and Software, Volume 80, Issue 7, 2007

Lingli Zhang, Chandra Krintz, and Sunil Soman, Efficient Support of Fine-grained Futures in Java, The 18th IASTED International Conference on Parallel and Distributed Computing AND Systems (PDCS), November 2006

Sunil Soman and Chandra Krintz, Efficient and General On-Stack Replacement for Aggressive Program Specialization, International Conference on Programming Languages and Compilers (PLC), June 2006

Sunil Soman, Laurent Daynès, and Chandra Krintz, Task-Aware Garbage Collection in a Multi-Tasking Virtual Machine, In Proceedings of the Internation Symposium on Memory Management (ISMM), June 2006

Sunil Soman, Chandra Krintz, and David F. Bacon, Dynamic Selection of Application-Specific Garbage Collectors, In Proceedings of the Internation Symposium on Memory Management (ISMM), October 2004

Sunil Soman, Chandra Krintz, and Giovanni Vigna, Detecting Malicious Java Code Using Virtual Machine Auditing, 12th USENIX Security Symposium, August 2003

Abstract

Memory Management for Multi-Application Managed Runtime Environments

Sunil Soman

Modern computing platforms are pervasive, networked, heterogeneous and increasingly complex. These systems range from small hand-held devices such as cell phones, to large servers that run back-end software. Portability, widespread use, and security of program execution as well as support for a fast program development cycle are key concerns of software developers and end users of these platforms. To address these concerns, modern programming languages and their implementations have emerged to facilitate high-level, object-oriented, and type-safe software development that is portable and secure. Popular examples of these languages are Java and the .NET languages. Programs written in these languages are encoded by a source compiler into an architecture-independent format, and then executed on any platform for which there is a virtual execution environment called a Managed Runtime Environment (MRE). MREs provide dynamic compilation of programs to the underlying native machine format, adaptive optimization, memory management, security verification, and other runtime services for programs.

Automatic memory management, or Garbage Collection (GC), is an MRE service that is key to facilitating programmer productivity, portability, and memory safety of

programs written in Java and the .NET languages. GC relieves the user from employing (and debugging) explicit deallocation of heap memory. However, since such management and reclamation is provided by the MRE, it necessarily introduces overhead. Much prior work has focused on mitigating the overhead of GC for MREs that execute a single program at a time using a single operating system process. However, today's MRE platforms execute a wide variety of applications with diverse computational characteristics, resource requirements and object lifetimes. Moreover, state-of-the-art MREs now are able to execute multiple applications using a single MRE process, serially or concurrently. For widespread use of such systems, and arguably for the success of such languages, we must advance GC technology to exploit the multi-program execution model and to enable efficient automatic memory management and program performance for the next-generation of portable software.

In this dissertation, we focus on memory management techniques for the next-generation of multi-program MREs for Java and consider both serial (persistent) and concurrent program execution models. In particular, we investigate, design, and engineer two novel solutions for GC that facilitate high-performance program execution for these MREs: (i) Application-Specific Garbage Collection and (ii) Scalable Memory Management for Multi-Tasking MREs. In the former, we customize and automatically specialize the GC in an MRE for a particular application and switch GCs automatically within a single, persistent MRE as needed for each new application

that the MRE executes. In the latter, we present a novel GC system for concurrent execution of programs; this system isolates GC activities of individual tasks while programs share a single heap. Our results show that we are able to achieve significant performance gains over the state-of-the-art MRE systems for both research and production-quality multi-program MRE systems in terms of throughput, response time, and memory footprint.

Contents

Acknowledgements **iv**

Curriculum Vitæ **v**

Abstract **vii**

List of Figures **xii**

1 Introduction **1**
1.1 The Thesis Question 7
1.1.1 Application-Specific Garbage Collection 7
1.1.2 Scalable Memory Management for Multi-tasking Managed Runtime Environments 10
1.1.3 Contributions 14
1.2 Outline 16

2 Background **17**
2.1 Garbage Collection for Managed Runtime Environments 17
2.2 Multi-Tasking Managed Runtime Environments 32

3 Application-Specific Garbage Collection **39**
3.1 Support for Garbage Collection Switching 43
3.1.1 Multiple Garbage Collectors in a Single JVM 44
3.1.2 Multi-purpose Object Header 57
3.2 Specialization Support for GC Switching 59
3.3 Annotation-Based Garbage Collector Selection 69
3.4 Automatic Garbage Collector Switching 73
3.5 Evaluation 75

3.5.1 Experimental Methodology 76
3.5.2 Results 77
3.6 Related Work 94
3.7 Summary 98

4 Scalable Memory Management for Multi-Tasking Managed Runtime Environments 100
4.1 Application-Aware Memory Management for Multi-Tasking Managed Runtime Environments 102
4.1.1 Hybrid generational heap 105
4.1.2 Per-application Old Generation Regions 109
4.1.3 Application-Concurrent Scavenging 117
4.1.4 Evaluation 120
4.2 Discussion 128
4.3 Scalable Hybrid Collection for Multi-Tasking Managed Runtime Environments 133
4.3.1 Hybrid Mark-Evacuate-Sweep Garbage Collector 137
4.3.2 Evaluation 146
4.4 Related Work 164
4.4.1 Application-Aware Memory Management 164
4.4.2 Scalable Hybrid Collection 168
4.5 Summary 172

5 Conclusion 175
5.1 Future Work 179

Bibliography 182

List of Figures

3.1 Performance using different GCs and heap sizes. The y-axis is total time in seconds. For SPECjbb, we report 10^6/throughput to maintain visual consistency. The x-axis is heap size relative to the minimum with the GC that can execute the program in smallest heap. 40
3.2 Performance using different GCs and heap sizes. The y-axis is total time in seconds. 41
3.3 Overview of our GC switching system. The JVM consists of the standard set of services as well as *multiple* GCs (an allocator and collector) as opposed to one per JVM image. The system employs the current GC through a reference to it called the CurrentPlan. When a switch occurs, the system updates the CurrentPlan pointer (and performs a collection in some cases). All future allocations and collections (if any) use the newly selected GC. 49
3.4 JikesRVM/JMTk class hierarchy: Original and switch-enabled. . . . 51
3.5 Virtual address space layout in the switching system (a) and a table (b) that indicates when a GC is required on a switch (from the row GC to the column GC) and its type: full (F), minor (M), or none (–). 53
3.6 Examples of bit positions in status word in object header 58
3.7 Shows how the VARMAP is maintained for a snippet of Java source (its bytecode and high-level intermediate representation (HIR) is included). We show the VARMAP entry for the `callme()` call site that contains the next bytecode index (25) after the call site `callme` and three local variables with types (`a: l8i, b: l15i, c: l17i`). 65
3.8 Shows how the VARMAP is updated after copy propagation. Variable `b: l15i` is replaced with `a: l8i`. 65
3.9 Additional inputs for SPECjbb2000 (in addition to input1 in Figure 3.1). 70

3.10 Additional input for SPECjbb2000 (in addition to input1 in Figure 3.1). 71
3.11 Inputs that we considered to evaluate GC behavior across heap sizes, the minimum heap size in which the program will run using our JVM, and the GC selection decisions with which we annotate each program to enable annotation-guided GC switching. 72
3.12 Performance of our OSR-VARMAP Implementation in JikesRVM Reference System. Figure shows the average execution time (excluding compilation) performance improvement enabled across heap sizes by our VARMAP implementation over using an extant implementation of OSR – a variation on the OSR points in JikesRVM. 79
3.13 Compilation overhead of our VARMAP implementation over the JikesRVM reference system. Columns 2 and 3 are compilation times in milliseconds and column 4 is the percent increase in compilation time. The final two columns show the compilation (collectable) and runtime space overhead, respectively, introduced by our system. 79
3.14 Overhead introduced by the garbage collection switching system when it never switches, over the clean (reference) JikesRVM. The percentage values are averaged over heap sizes. On average, the GC switching system adds a 15% overhead over the clean JikesRVM, *when no switching is triggered*, due to support for on-stack replacement. 80
3.15 The overhead introduced by the VARMAP version of the GC Switching System over a clean system without GC switching functionality. By reducing the overhead of the Orig-OSR implementation, we are able to cut the base overhead of the GC switching system (the overhead imposed when the system *does not switch*) from 15% to 5%, i.e. the resulting version of the system introduces 5% base overhead over the clean system. 81
3.16 Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows two examples with switch points. 83
3.17 Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows an example with switch points. 84
3.18 Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows two examples without switch points. 85

3.19 Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows an example without switch points. 86
3.20 Summarized performance differences between our annotation-guided switching system and the reference system for small heap sizes (minimum for an application to 3x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst performing GCs across small heap sizes (the time in milliseconds that this equates to is shown in parenthesis). 87
3.21 Summarized performance differences between our annotation-guided switching system and the reference system for medium to large heap sizes (from 3x the minimum for an application to 8x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst performing GCs across medium to large heap sizes (the time in milliseconds that this equates to is shown in parenthesis). 88
3.22 Percent degradation of our system over the widely used GMS collection. The negative values indicate that on average across heap sizes, our system improves performance over GMS. 90
3.23 Performance of automatic switching when memory resources are suddenly constrained. Columns 2 and 3 show the time in seconds for execution for the clean (Base) system and our automatic switching system (including all overheads). Column 4 shows the percent improvement enabled by our system. The right half of the table shows the OSR statistics: number of OSRs, total OSR time in milliseconds, and the heap size following the memory reclamation by the system. 94

4.1 Application independent flexible young generations. A generation virtualizer maps applications to young generations. Each generation comprises one or more eden spaces, each of which consists of an integral number of contiguous chunks allocated from a pool. Eden spaces of an application are linked together. Chunks can be added or removed dynamically. 108
4.2 Example of region management & tenured space reclamation at application termination without a full GC. (A) Initial configuration. (B) Both applications 1 & 2 have performed promotions and their respective full region list are now non-empty. (C) Application 1 terminates and its set of full regions is added to the global free list. (D) Application 3 enters the system and application 2 & 3 start using space allocated from the region free list. 113

4.3 Example illustrating shrinking of old generation footprint upon application termination. 114
4.4 Region adjustment at full GC. *pa* is the region to be adjusted. 116
4.5 Per-application safepointing; *begin_per_application_safepoint* initiates a safepoint for a single application and *end_per_application_safepoint* ends it and resumes mutators for that application. 119
4.6 Throughput improvement enabled by independent young generations & regions for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications: `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program. 121
4.7 Response time improvement enabled by independent young generations & regions for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications – `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program. 122
4.8 Data for the Base MVM system (shared new generation). Columns 2 & 3 show the number of minor (scavenges) and major collections respectively for a single instance of the benchmark in Column 1. The rest of the columns show execution time (ET) in seconds & GC time (GCT) in milliseconds for 1, 2, 3, 4 and 5 concurrent instances, respectively, of the programs listed. Figures 4.9 and 4.11 show improvement relative to this data. 123
4.9 Total end-to-end performance improvement enabled by mutator-concurrent scavenging over the base MVM for homogeneous benchmark instances. Bars indicate increasing number of applications (from 1 to 5). 125
4.10 Change in the number of GCs (minor and major) with mutator-concurrent scavenging over the base MVM for 1 thru 5 instances of the same benchmark. 125
4.11 Total GC time improvement (minor + major) enabled by mutator-concurrent scavenging over the base MVM. Bars indicate increasing number of homogeneous applications (from 1 to 5). 126

4.12 Performance of a state-of-the-art multi-tasking MRE (MVM) with per-application young generation GC versus multiple instances of the Java HotSpot virtual machine for *concurrent* execution of five community benchmarks. No prior work has performed such an evaluation. Although per-application young generation GC significantly improves performance over prior state-of-the-art, for programs that involve significant old generation GC activity, performance suffers due to the choice of an unsuitable old generation GC algorithm. 130
4.13 Marking, Evacuation and Sweeping of Old Regions. Each application has a corresponding list of live areas. Marking traverses live objects for an application and marks live objects in the mark bitmap. After marking, candidate regions for evacuation (or sweeping) are selected based on the amount of live data and fragmentation. Regions selected for evacuation are then evacuated, regions selected for sweeping are swept and free areas in these added to a per-application free list. Pointer adjustment for swept regions is also performed during this pass, if necessary. 143
4.14 Adjustment of old regions. Application 1 is being collected. We build the region connectivity matrix for application 1 during the marking phase. Region 2 has outgoing pointers to Region 3, therefore, Region 2 must be scanned if Region 3 is evacuated. However, Region 1 and 4 do not must be scanned. 145
4.15 Benchmarks and workloads used in the empirical evaluation of MTM 147
4.16 Total execution time (in seconds) and footprint (in MB) data for MTM with application-aware memory management and hybrid old generation GC for concurrent homogeneous (multiple instances of same application), and heterogeneous (multiple instances of different applications). The benchmarks are described in Figure 4.15. All relative performance improvement results for execution time as well footprint in this section use these values. 148
4.17 Percentage improvement in execution time enabled by MTM (MVM extended with per-application hybrid GC) with per-application hybrid GC versus a prior implementation of MVM (cf. Section 4.1) when executing concurrent workloads that show significant old generation GC activity. MTM enables better performance due to a more efficient old generation GC and performance isolation. 150

4.18 Old generation GC times (total) for MTM (MVM extended with per-application hybrid GC) versus a prior implementation of MVM described in Section 4.1. GC times are presented in seconds along with percentage improvement in GC time enabled by MTM . MTM 's per-application hybrid old generation GC outperforms mark-compact old generation GC used in the prior implementation. 151
4.19 Percentage improvement in execution time enabled by MTM over HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (multiple instances of the same application). Benchmarks are described in Figure 4.15. 154
4.20 Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.15. 154
4.21 Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application). 155
4.22 Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix. 156
4.23 Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = 32MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application). 157
4.24 Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = 32MB), heterogeneous workloads, i.e., multiple concurrent instances of different applications. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix. 158
4.25 Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = 32MB) homogeneous concurrent workloads. Benchmarks are described in Figure 4.15. 159
4.26 Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = 32MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.15. 159

4.27 Percentage improvement in execution time enabled by MTM over HSVM for 1 through 4 times the minimum heap size that each benchmark needs to execute in MTM 160
4.28 Footprint for MTM with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for a set of homogeneous (instances of the same application) and heterogeneous (different applications) concurrent workloads. Hybrid GC achieves a footprint that is lower than always choosing mark-sweep or always choosing copying. 162
4.29 Execution time MTM with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for the `javac` benchmark. 163

Chapter 1

Introduction

Computing platforms today are heterogeneous, pervasive and networked, and range from small devices such as cell phones to large-scale server farms. These systems are highly complex as well as diverse in their architectures, resource constraints, and capability. To provide a development infrastructure for modern systems that facilitates programmer productivity across a wide range of systems, programming language technology, such as that for Java and the Microsoft .Net framework, has emerged to provide high-level, object-oriented, type-safe, secure, and portable application development.

These language technologies implement a ”write once, run anywhere” programming model in which the source program encoded in an architecture-independent binary format is executed (after potentially being transferred over a network) on a machine with a *Managed Runtime Environment (MRE)*. An MRE is an execution environment that virtualizes the underlying hardware and resources for programs, verifies

that programs are well formed and type-safe, provides dynamic loading and library support, and that implements a wide array of runtime services. Such services include garbage collection, dynamic compilation and optimization, thread management, and incremental program (class) loading. This execution model precludes the need for programmers to have expert knowledge of the vast array of underlying architectures, platform-specific resources, and complex program behavior, since a program, without modification, can execute on any system for which there is an appropriate MRE. MREs thus facilitate significant programmer productivity as well as program portability and security. Examples of popular MREs include the Java Virtual Machine (JVM) [71], and the Microsoft .NET common language runtime (CLR) [17].

Modern MREs typically execute a wide variety of programs, ranging from scripting languages, bytecode compilers, GUI programs to database applications and application servers. These programs have diverse resource requirements (CPU, memory, network, disk), differences in the number, size and lifetimes of dynamically allocated objects, and different execution characteristics (differences in compiled code, single or multi-threaded execution).

There also exist differences in the execution model for these applications. A *single-tasking* MRE executes a single application in a given instance of the MRE, i.e., an operating system (OS) process is spawned for each application. Single-tasking MREs rely on the underlying operating system (i.e. the process model) to isolate

programs from each other for security, as well as for resource management and accounting.

With the widespread use of Internet computing, users execute multiple, diverse applications on a single platform. Instantiating an MRE every time a new application is deployed adds startup and execution time overhead. Recent MRE advances thus, now provide support for *persistent execution* [100, 4], which precludes the need to terminate an MRE instance upon application termination. That is, the lifetime of the MRE exceeds that of any single application. With this persistent model of execution, there exist opportunities for the MRE system to learn from past execution and behavior and to adapt the MRE services (and program behavior through compilation) over time to extract high performance from applications.

Further, as desktop and hand-held platforms become more capable (faster multicore CPUs, larger memories, etc.), users that once executed a single program at a time, now demand that these systems *multi-task*, i.e., seamlessly and simultaneously execute multiple applications (such as, instant messaging, calendar and email clients, audio player, Internet browsers, office suite, etc.). Single-tasking persistent MREs duplicate effort across MRE instances, MRE services, internal representations of classes, code, etc., cannot be shared across programs. Such redundancy increases startup time and memory consumption and degrades overall system performance and scalability. A *multitasking, persistent, MRE* implementation [29] enables sharing of

common and scarce resources between applications while maintaining portability, mobility, and type-safety.

To achieve safe, flexible, portable, and efficient execution, MREs provide

- Dynamic class loading. The ability to load, link and unload library and application classes at *execution time* (runtime).
- Type-safe class file verification. To guarantee secure execution, type-safety is checked when classes are loaded.
- Dynamic compilation, i.e., application and library code is compiled *on-the-fly* at execution time.
- Adaptive application optimization. Since code is dynamically compiled, the execution time of an application includes the time required to compile application and library methods. Adaptive optimization aims at expending most effort on critical methods and code regions. Most time is spent on compiling "hot", or frequently executing code (more extensive and time consuming optimizations), whereas "cold" or less frequently executing code is optimized less aggressively or not optimized at all.
- Automatic memory management. To improve programmer productivity and memory safety, MREs implement dynamic, automatic reclamation of explicit heap allocated data, i.e., garbage collection (GC).

However, the services provided by an MRE *execute at application execution time* and necessarily impose overhead on the programs. This overhead is experienced by the user as decreased program responsiveness and increased overall execution time.

A key MRE component that significantly impacts program performance is garbage collection, i.e., automatic memory management. Heap memory used by applications is not explicitly freed by programmers in source code. The garbage collection system identifies and recycles unreachable heap objects automatically for the program. Garbage collection, therefore, can be potentially disruptive since it consumes CPU cycles that would otherwise be available to the application. Prior work has shown that garbage collection can impose significant overhead on both application execution and response time [18, 1, 44].

Modern garbage collection techniques [79, 43, 14, 52, 16, 9] address the overhead imposed by basic garbage collection systems [97]. However, modern applications have evolved to be very complex in terms of both their implementation and their dynamic behavior, precluding any one MRE memory management approach to provide high-performance for all applications (due to their different resource requirements and usage patterns). An MRE optimized to perform well for one type or class of applications, may inhibit performance for other types of applications. Prior work has noted that the application performance is significantly impacted by the choice of the GC algorithm [5, 40]. State-of-the-art MREs provide support for multiple GC

algorithms, however, the GC algorithm must be specified by the user when the MRE is instantiated, i.e., GC selection is *not dynamic*. The MRE must be terminated and restarted with an appropriate GC for a new application, which adds startup overhead and which also means that commonly executed code must be recompiled.

Further, the lack of cooperation between applications running within the same MRE (multi-tasking) and the resource contention between MREs (single-tasking) on the same system can lead to additional performance degradation. For example, if there are multiple applications executing on a single platform (e.g. through a web browser, or by a user executing multiple programs) *simultaneously*, garbage collection strategies may interfere with each other impacting the entire system. Memory management in state-of-the-art multi-tasking MREs inhibit application performance. To facilitate sharing, multiple applications execute in the same address space [29]. Consequently, garbage collection triggered by any application degrades performance for all other applications. In addition, resources freed by an application on termination cannot be readily reused by other applications without disruptive GC activity. Moreover, precise resource accounting and tracking is not possible. The performance degradation due to GC is exacerbated as additional applications execute simultaneously, since scalability is severely inhibited.

1.1 The Thesis Question

The question we attempt to answer in this dissertation is the following,

How can we achieve high-performance memory management in Managed Runtime Environments that execute multiple applications?

We investigate the effect of the choice of garbage collection policies on the performance of different applications executing serially (one after the other) in an MRE. We demonstrate that an *application-specific* GC policy that is better suited to a particular application enables significantly better performance for that application, compared to selecting a single generic GC for all applications in a single-tasking persistent MRE (henceforth, referred to as a single-tasking MRE).

We then consider the performance of multiple concurrent applications executing in a single persistent MRE instance. This execution model raises several questions about resource management and accounting, sharing, performance isolation between applications, and footprint size, which we investigate in this dissertation. An next overview the two foci of this dissertation.

1.1.1 Application-Specific Garbage Collection

Most MREs [59, 91] use general-purpose GC algorithms that attempt to enable high-performance execution across all applications. However, prior research [5, 40,

101, 82], has shown that the efficacy of a memory management system (the allocator and the garbage collector) is dependent upon application behavior and available resources. That is, no single collection system enables the best performance for all applications and all heap sizes. Our empirical experimentation confirms these findings in a performance-oriented, server-based, Java virtual machine, JikesRVM [2] from the IBM T.J. Watson Research Center. Over a wide-range of heap sizes and the 10 benchmarks studied, we found that *every* collector enabled the best performance at least once; *including* a mark-sweep and non-generational copying collector, two collectors that are commonly thought of as implementing obsolete technology. We hypothesize that to achieve the best performance, the collection and allocation algorithms used should be selected according to both application behavior and heap size. Currently, such selection can only be done by the user and is very challenging to get right given the performance range of different garbage collectors for different heap sizes.

Existing execution environments enable application- and heap-specific garbage collection, through the use of different configurations (via separate builds or command-line options) of the execution environment. However, this methodology for GC selection, in addition to being challenging to get right, is not amenable to next-generation, high-performance server systems in which a single execution environment executes

continuously (persistently) while multiple applications and code components are uploaded by users [56, 75, 10]

For persistent MREs, a single collector and allocator must be used for a wide range of available heap sizes and applications, e.g., e-commerce, agent-based, distributed, collaborative, etc. As such, it may not be possible to achieve high-performance in all cases and selection of the *wrong* GC system may result in significant performance degradation. To address this limitation, we present the design, implementation, and evaluation of a dynamic GC switching system for JikesRVM. Our switching system facilitates the use of the garbage collector and memory allocator that will enable the best performance for the executing application *and* the underlying resource availability. The system we present is extensible and general; it can switch between many different types of collectors, e.g., semi-space, mark-sweep, copying-mark-sweep, and many variants of generational collection.

To evaluate our system, we have implemented two mechanisms: annotation-guided GC selection, and automatic switching. For the former, we identify the best performing GC for a range of heap sizes for each program, across inputs. We then annotate the programs to identify the collection system to use for a range of available resource levels. Upon dynamic loading of each application, the MRE uses the annotation to switch to the appropriate GC given the current maximum available heap size. To implement automatic switching, we employ a simple heuristic that uses maximum

heap size and a heap residency threshold to switch during program execution. The second part of this dissertation focuses on memory management for multi-tasking MREs.

1.1.2 Scalable Memory Management for Multi-tasking Managed Runtime Environments

Multi-tasking MREs execute multiple, isolated applications in a single MRE instance that is persistent. Co-locating programs in the same address space simplifies the virtual machine implementation through sharing of the runtime representation of programs and dynamically compiled code. Such sharing also avoids duplicated effort across programs (e.g. loading, verification) and amortizes runtime costs, such as dynamic compilation, over multiple program instances. Prior work on the MVM [27], shows how a multitasking design reduces startup time and memory footprint, and improves performance over a single-program MRE approach.

Multi-tasking MREs provide isolation and resource management for multi-application workloads and provide application developers with a first-class representation of an isolated program execution (e.g., the *isolate* in [57, 29] and the *application domain* in *.Net* [63]). This representation provides the necessary functionality to launch and control the life cycle of multiple, isolated execution units (programs).

MREs have access to high-level program information, can monitor time-varying program behavior and underlying resource availability, and can dynamically optimize programs as well as the runtime based on prior information. Therefore, they potential for more intelligent scheduling and resource management of programs. Prior work has shown that multi-tasking is more effective at enabling cross-program sharing of dynamically loaded and compiled code, and at achieving smaller memory footprint and faster startup times [28, 30] than traditional MREs that rely on process-based isolation. Yet, little attention has been directed at the *performance of multi-tasking MREs for simultaneous program execution*, i.e., concurrent workloads, compared to a more common scenario in which each program runs in its own process.

Application diversity and widely varying resource requirements implies that applications may interfere with the execution of other applications within the same MRE. A MVM GC implementation must address unique challenges not faced by GC systems in single-tasking MREs to achieve scalable performance, such as,

- Each application that executes in the MRE instance must not interfere with other programs, either functionally, or in terms of performance. In particular, a GC triggered by any application should not impact the performance of other applications.

- An application should have control over heap and GC system parameters, such sizing or generational tenuring parameters.

- Heap resources reclaimed from applications should be made available to other applications if required.

- Upon application termination, heap resources that the application has allocated must be immediately reclaimable and available for use by other applications, without the need for expensive GC operations. In addition, such reclamation should not adversely affect other applications.

- Scalability should be guaranteed, i.e., when multiple concurrent applications are executed, the memory management subsystem should scale and not introduce overhead that is proportional to the number of applications executing.

- When running applications concurrently, footprint must be restricted so that a multi-tasking MRE does not consume more memory compared to multiple instances of single-tasking MREs, *while preserving the performance benefit due to sharing*.

We present an MRE memory management design that addresses these challenges for the Sun Microsystems MVM [92] for Java. Key to our design is

- An organization of the heap that enables per-application performance isolation for the memory management system
- Independent and per-application allocation and collection of objects
- A GC technique that provides high throughput and scalability for concurrent application workloads, *while* ensuring that heap footprint is restricted.
- GC-free memory reclamation upon application termination.

The system we propose, is the Multi-Tasking Memory Manager (MTM), which leverages two novel techniques – (i) Application-aware memory management, i.e., ensuring that heap management, garbage collection and re-allocation is cognizant of the fact that multiple applications might execute concurrently in a single MRE instance. Application-aware memory management provides heap memory isolation for each application and enables any application to allocate and trigger GC independently of other applications executing in the MRE instance. (ii) Scalable hybrid GC that ensures that a low footprint in maintained for the MRE, while achieving high throughput by providing a combination of two different garbage collection techniques, mark-sweep collection and copying collection for longer-lived objects allocated by the application. We implement MTMin the MVM (an extension of the production-quality HotSpot MRE from Sun Microsystems) and compare our contributions against the state-of-the-art in multi-tasking MRE technologies as well as

against a single-tasking HotSpot system, for a wide range of community benchmark programs and multi-tasking scenarios.

1.1.3 Contributions

In summary, with this dissertation we contribute,

- An evaluation of the effect of different GC algorithms on application performance using a single-tasking MRE.
- The design and implementation of a novel framework that enables the garbage collector and allocator to be switched dynamically in a single-tasking MRE, at runtime, i.e. while the MRE and the application are executing.
- A general-purpose on-stack replacement mechanism that extends prior work by allowing the ability to perform on-stack replacement without the need to insert special instructions and checks (guards) into the application code. Our on-stack replacement mechanism allows code that has been specialized for a specific GC to be de-specialized when a GC switch is performed (if necessary).
- Two techniques that employ our switching framework to dynamically switch GCs – annotation-guided switching that uses program annotations, and adaptive switching that uses online heuristics based on the amount of time spent in

GC versus the amount of time spent executing application code, and the frequency of GC.

- An empirical evaluation that shows the efficacy of GC switching for annotation-guided and adaptive switching over manual selection of GC system.

- The design and implementation of a novel memory management system for multi-tasking MREs, in particular, for the Sun Microsystem Labs' state-of-the-art Multi-tasking Virtual Machine. Our advances include a heap layout that enables applications to allocate memory on demand in a shared address space and track heap resources precisely, a synchronization mechanism that enables an individual thread to be pauses for GC (instead of all threads), per-application young generation GC that can execute concurrently with threads of other applications, reclamation of heap space used by an application on its termination without requiring GC.

- The design and implementation of a novel hybrid and adaptive old-generation GC technique for multi-tasking that combines two different GC algorithms, mark-sweep and copying GC, and uses online heuristics to apply either mark-sweep or copying to different parts of an application. This technique enables high performance for concurrent workloads that significantly exercise the old generation as well as maintains good process footprint.

- An evaluation of a multi-tasking MRE that compares our techniques against its single-tasking counterpart for concurrent workloads. Our results indicate that with a carefully designed memory management system, multi-tasking can perform to its potential and enable significantly better performance compared to single-tasking.

1.2 Outline

The outline of the remainder of this dissertation is as follows. In Chapter 2, we provide an overview of memory management techniques in modern Managed Runtime Environments, followed by a discussion of prior work related to application-specific garbage collection. We then describe multi-tasking MREs that are capable of executing multiple applications in isolation in a single operating system instance.

In Chapter 3, we describe our work on application-specific garbage collection for single-tasking MREs. We present evidence that shows that general-purpose GC may not best suited for all applications and resource constraints. We then present our application-specific GC framework that allows dynamic selection of and switching between diverse GCs, and present two applications of the framework.

The second part of the dissertation 4 details scalable memory management for multi-tasking MREs, including our extensions to a state-of-the-art MRE for application-

aware memory management and scalable, hybrid GC. In Chapter 5, we present our conclusions and plans for future work.

Chapter 2

Background

In this chapter, we provide background on garbage collection in MREs. We discuss basic GC terminology, followed by commonly used GC techniques in MREs to which we refer in the rest of this dissertation. We then provide background on multi-tasking MREs, the foci of the second part of this dissertation (Chapter 4).

2.1 Garbage Collection for Managed Runtime Environments

In this section we will define garbage collection terms and examine commonly used collection techniques to which we refer in the remainder of this dissertation.

Garbage

Garbage is defined as data allocated dynamically by a program that is no longer reachable.

Mutator

In garbage collection terminology, an application thread is called the mutator, since it mutates or modifies objects.

Garbage Collector

The term *garbage collector* signifies an automatic memory *reclamation* mechanism, but the data structures for memory management are shared by the reclamation mechanism and the *allocator* [69]. The choice of the allocation algorithm is generally tied to choice of the reclamation mechanism. The allocator and the collector can, therefore, be implicitly considered to be two components of a garbage collector.

Garbage Collection Cycle

A garbage reclamation algorithm consists two phases – *garbage detection*, in which live objects are distinguished from garbage, and *garbage reclamation*, in which the space occupied by garbage is freed for use by the application program (the *mutator*). Detection and reclamation constitute a garbage collection cycle.

Root Set

To detect live objects, a *root set* of references is used. This consists of an application's static variables, any local (stack-allocated) variables, and general-purpose registers. All objects that are directly or transitively *reachable* from the root set are assumed to be live and cannot reclaimed. Objects that are unreachable are considered to be garbage, and therefore can be recycled. *Reachability* is a more conservative approach than one used in, say, an optimizing compiler, which considers variables to be dead if they are *unused* after a certain point in the program.

Object Header

Modern object oriented languages are dynamically typed and objects typically have meta-data space to hold type information. MREs typically make use of a single, two- or three-word object header. The header usually contains an object's identity hash code (a unique identifier) and GC status information, the format of which is MRE dependent. In addition, it also contains a reference to the object's class type, which is typically implemented as an object itself. All objects of a certain type refer to the class object.

Compiler Support for Garbage Collection

Modern programming languages for virtual machine applications are strongly typed, which means that the compiler fully supports runtime type identification. Consequently, it is possible to accurately identify object references without any need for a conservative approach. A conservative garbage collector is used in languages that have do not provide strong typing, e.g. C and C++. A conservative garbage collector does not know the location of all object references inside an object; such a collector must be *conservative* in its identification of object references, i.e., anything that *looks* like a pointer, may be one. On the other hand, garbage collectors designed for modern MREs have full knowledge about an object's type and its internal reference fields, and are therefore, fully *type accurate*.

Interpreters and compilers differ over when garbage collection can be initiated. GC can be initiated at any point during program execution in case of interpreters, called *GC-anytime*. However, all modern execution environments employ an optimizing compiler, usually with multiple optimization levels. In such a case, garbage collection can generally only be initiated at certain defined points during execution, called *safe-point* collection. This allows the optimizing compiler to use complex optimizations between safe points, as long as it maintains information that is necessary to locate pointer values at safe points. This information is generated during compi-

lation and is maintained in a per-method data structure called the *garbage collection map* (GC map). We will next discuss commonly used garbage collection algorithms.

Mark-Sweep Collection

Mark-sweep GC consists of two phases.

- Mark phase. Objects that are directly or transitively reachable from the root set are marked live. Marking can be done by setting a bit in the object header or in a global bitmap data structure.

- Sweep phase. In this phase, objects that are not marked in the mark phase are swept or reclaimed for use by the mutator. Freed objects are linked into a free-list maintained by the heap allocator. Allocating from a free-list, however, is more expensive than allocating from a contiguous memory region.

The mark-sweep technique is able to reclaim garbage cycles, and it imposes no overhead during object manipulation, however, some amount of work is required at object creation time, to initialize the object header (if the mark bit is stored in the header). Mark-sweep GC, however, has some drawbacks.

- Fragmentation. Since reclamation is done in place, in time, free areas are interspersed with live objects, leading to a fragmented heap. Object allocation might fail even though the total amount of free space is sufficient to honor

the allocation request. Various techniques are used to mitigate this problem, e.g., maintaining free lists with different block sizes called segregated lists, and buddy lists in which blocks from adjacent lists are coalesced [98].

- Collection cost. The collection cost of mark-sweep collection is proportional to the size of the heap. While reclaiming objects during the sweep phase, garbage as well as live objects are visited. This cost can be mitigated by using a bitmap. Typically, a bit in the bitmap corresponds to a fixed number of bytes in the heap. The cost of traversing the bitmap is less than that of scanning the heap.

- Poor Locality of Reference. Since objects are allocated and freed in place, freshly created objects may be spatially closer to older objects, leading to poor locality of reference if objects are accessed in allocation order.

Mark-Compact Collection

To handle fragmentation, poor locality, and expensive allocation associated with mark-sweep collection, mark-compact collection was invented [25]. In mark-compact collection, the initial marking phase is similar to the marking phase of mark-sweep collection, however, the reclamation phase attempts to compact live data into one contiguous region of the heap. This solves the fragmentation problem. As a result of compacting live data, free space exists as a contiguous region. Consequently, allo-

cation is inexpensive as it involves only a pointer increment into the contiguous free space, which is also known as *bump pointer* allocation. Locality is also improved, since objects of a similar age are clustered together in space.

The simplest compacting reclamation can be thought of as a *sliding* compaction that "slides" live objects into spaces left behind by dead objects. However, the actual process of compaction can be quite expensive, and requires multiple passes: one pass to identify the new locations for the live objects, another to compute new locations for live objects, and a third pass to adjust references to objects that have moved.

Copying Garbage Collection

Compaction is an inherent part of copying garbage collection, in which all live data is copied to one part of the heap, so that it is contiguously laid out. The rest of the heap is then considered to be free, and can be used by the allocator for future allocations. Copying collection is often considered to be *implicit*, or scavenging, since garbage is not explicitly identified and reclaimed.

The heap is usually divided into two equal parts or *semispaces*: the *from space* and the *to space*. All allocation is from the from space, which is the considered to be the "current" semispace. The to space is always empty while the application executes. During a collection, live data is copied from the from space to the to space. The from space now contains only garbage and can be reclaimed. This is done by swapping the

roles of the two spaces (usually by merely updating references to the from space and the to space). Thus, at the end of the collection cycle, the to space is empty and live data has been contiguously arranged in the from space.

Cheney's traversal algorithm [23] is the most popular method for identifying live data. As mentioned before, the *root set* for a garbage collection consists of static variables, stack variables, and register references. The objects in this set are first added to a queue, which is then scanned in a breadth-first manner to identify objects that are reachable from the root set. These objects are in turn added to the queue, in order to identify heap objects reachable from them, and this recursive process continues until all live objects have been traced. Every object that is identified as live is copied to the to space, and a *forwarding bit*, is set in the object's header. Along with the forwarding bit, a forwarding pointer is also stored, which indicates the new location of the object. The forwarding pointer enables updates to all pointers that refer to the object. The use of the forwarding bit avoids duplicate copying of live objects and guarantees termination.

Copying collection has several advantages. Similar to the mark-compact technique, it employs fast bump-pointer allocation, which can be implemented as a simple increment into the free space (usually, along with a boundary check, to decide whether a garbage collection should be initiated). Fragmentation is non-existent since the algorithm is inherently compacting. Locality, however, may not always

be improved – Cheney's breadth-first traversal may not preserve locality of reference. Depth-first traversal has been used in some recent implementations [89].

Unlike mark-sweep and its variation, the mark-compact algorithm, the amount of work during copying garbage collection is proportional to the amount of live data, and not to the size of the heap. In addition, unlike mark-compact collection, a single pass over the live data is sufficient. Mutator overhead is similar to mark-sweep collection, since during object creation, the forwarding pointer and the forwarding bit have to be initialized.

The biggest disadvantage of semispace copying collection is that the heap area available to an application is only half of the entire heap space. An application's memory requirement is doubled compared to mark-sweep or mark-compact collection. Another major disadvantage is that the process of reclamation is quite slow, given the need to copy every single live object. However, since the bump pointer allocation is a feature of copying collection, this algorithm performs quite well when garbage collection cycles are infrequent.

Generational Garbage Collection

As noted previously, mark-sweep collection needs to scan the entire heap space, a process that can be quite expensive. Mark-compact collection performs a number of passes over the entire heap, and if the amount of live data is high, performance

suffers. Copying collection also has high overhead, particularly if the amount of live data is proportional to the size of a semispace.

Dynamically allocated objects have been shown to follow the *weak generational hypothesis* [70, 94] that states that most dynamically allocated objects (between 80 to 90%) have very short lifetimes, and only a small percentage of objects live much longer. Generational garbage collection exploits this property. Objects that have been recently created (also called *young* objects), are segregated into a *nursery* (young object) area. As objects age in the heap (usually indicated by their survival after one or more garbage collection cycles), they are promoted (copied) to another area of the heap, called the *mature space*. The promoted objects are called *old* or *mature* objects.

The basic idea behind this approach is that the garbage collector should not have to process (either mark, or copy) objects that are going to survive well into the application's lifetime. Effort can be concentrated on collecting young objects, which will die sooner. The process of collecting the nursery area is called a *minor collection*. The mature space should also be examined once in a while to check whether older objects have become garbage. This is called a *major collection*, and is done less frequently. New objects are allocated only from the nursery. Any collection algorithm could be used to collect the nursery and the mature space, but promotions always *copy* live data from the nursery.

The generational technique leads to reduced garbage collection overhead, since excessive processing of live data is eliminated. This comes at a price – to perform minor collections without collecting the mature space, some bookkeeping is required. It is necessary to keep track of references from the mature space to the nursery, otherwise objects in the nursery that are referenced by objects in the mature space will be incorrectly reclaimed as garbage. This is done by means of a *write barrier*, which is a conditional or an unconditional check inserted into the compiled code in order to track pointer stores. The objects in the mature space that reference nursery objects must be remembered and included in the root set of objects for minor collection.

References from old to young objects must be remembered, so that a minor collection may occur independently of a major collection. Old objects that reference young objects are included in the root set for minor collection. Write barriers are used to keep track of pointer stores.

There are different write barrier techniques that differ mainly in the granularity of information stored. Hosking et al [50] compared these three techniques in a virtual machine with a Smalltalk interpreter. The three mechanisms they considered are: *remembered sets*, *card marking*, and *page protection*.

Remembered sets are the most accurate of the three, since they record the actual old object (or the slot containing the object) that references a young object. Hosking et al implemented remembered sets using two alternative implementations. A hash

table is the most standard way of storing remembered set entries, however, insertion overhead might be considerable. Consequently, they introduced the concept of a *sequential store buffer* (SSB), which consists of pages arranged contiguously and bounded by a limit or guard page. This allows the use of a simple pointer increment-and-store operation to remember entries. An attempt to store into the SSB past its limit is trapped by an operating system trap, which can be handled by the runtime.

Another scheme for remembering cross-generational stores is card marking, in which the heap is divided into multiple cards, with every card represented by a unique entry in a *card table*. The source card that contains the old object is marked, instead of remembering the object itself. Thus the level of granularity is much coarser. This implies that the pointer store check can be performed quickly, but at the cost of garbage collection time overhead, since the entire card has to be scanned to locate all references to nursery objects. The card marking scheme as originally introduced by Wilson [99] made use of a *bit* per card. The authors used a *byte* per card, which makes the process of checking and marking more efficient, since the smallest unit of memory access on most architectures is a byte,

Hoelzle [45] noted that the pointer store check in Wilson's basic card marking scheme [99] is quite slow, since a bit vector must be read from memory, updated, and then written back. Chambers et al [21] tried to improve on this by using a byte per card, instead of a single bit. On most architectures, marking a byte is much faster than

marking a bit – on a SPARC, this process can be performed in 3 instructions. Sun Microsystems' HotSpot VM [91] uses byte marking when updating the card table on mutations.

Hoelzle, attempted to further reduce the pointer store check overhead, by reducing the three-instruction write barrier to a two-instruction write barrier. This is significant, since he also demonstrated that pointer store checks constitute about half of the performance overhead associated with card marking (the other half is the time taken to scan the card table during minor garbage collection).

Hoelzle used a relaxed card marking scheme that uses an approximation during the card marking process, at the cost of some additional overhead during minor garbage collection. According to this scheme, a card containing an old-to-young reference is not remembered precisely, but rather, an entry in the card table corresponds to more than one card. This approximation saves one instruction per pointer store, compared to Chambers et al's accurate card marking. However, the extra scanning overhead due to this approximation might be too large for large objects and arrays. For such objects, Hoelzle used accurate card marking. The overhead due to store checks is determined by running benchmark programs with an instruction-level simulator, which models the exact hardware behavior, including cache behavior.

Blackburn et al divided the pointer store check into two parts – a *fast path* which performs the check to determine whether the reference is from an old to a young

object, and a *slow path*, which actually inserts the object (or its slot) into the SSB. The fast paths for the two cases, viz., remembering the old object or remembering the slot, are different. For the former (remembering the object), the old object's header word is checked for the presence of an *OBJECT_BARRIER* bit. If the bit has not been set, the slow path is taken, which will set the bit as part of the process of remembering the object. The *OBJECT_BARRIER* bit is cleared for every object when it is first created. Since, new objects are only allocated from the young space, correctness is ensured. For the case in which the *slot* containing the object is to be remembered, the fast path is implemented using a technique by Stefanovic et al [87]. The young object space is located in high memory and the old object space is in lower memory, with both spaces aligned on a boundary (2^k). Consequently, a simple bit-mask-and-shift can be used instead of an `and` operation. This technique is used in the Jikes Research Virtual Machine [13] that we use for implementing the techniques in the first part of this dissertation.

With a fully inlined write barrier fast, as well as slow paths are inlined at the site of the pointer store. A partially inlined check inlines only the fast path, and an out-of-line check makes use of a direct functional call without any inlining. The authors measured the compilation time for the three implementations by fully compiling their benchmark programs using the JikesRVM's optimizing compiler [19]. They also measured application performance without considering compilation time. The

compilation measurements show that full inlining incurs a heavy compilation time penalty (up to 25% worse compared to partial inlining, and 38% worse compared to the out-of-line check). The out-of-line check and store has the least compilation time overhead. The authors then showed that the slow path is rarely taken for their set of benchmark programs (0.15 to 3%). This implies that the fully inlined write barrier will not improve application performance by a noticeable amount. In fact, the authors reported that full inlining actually *degrades* application performance. This is probably due to poor locality and register allocator performance, as a consequence of excess code generated at pointer store sites. The out-of-line write barrier performs worse, since an out-of-line function has to be invoked for every pointer store. Partial inlining enables the best application performance. The authors also showed that the partially inlined slot barrier (in which the remembered set holds slots that contain the old object) performs better than the partially inlined object barrier (the remembered set holds the actual old objects). This is probably due to the fact that for the object barrier, the collector must scan the stored old object for pointers from the mature space to the nursery. The slot barrier remembers more pointers and a scan of the old object is not necessary.

We next discuss prior work on multi-tasking MREs, which the second part of this dissertation focuses on.

2.2 Multi-Tasking Managed Runtime Environments

Modern type-safe programming languages rely on an execution environment that can provide protection, security through isolation between applications, secure communication and resource management and accounting.

Typically, application safety, isolation, communication, resource accounting and management are provided at the operating system (OS) level. Each application executes in its own MRE instance, i.e., an OS process is spawned per application.

However, launching a separate MRE instance for each application is wasteful, since each MRE instance has a non-trivial base memory footprint *even when no application is executing* [7]. In addition, initiating a new MRE instance incurs a startup delay.

This execution model duplicates effort across MRE instances, since it prohibits sharing of MRE services and internal data structures across programs. Such redundancy increases startup time and memory consumption and degrades overall system performance and scalability.

Consequently, multi-tasking MREs have been proposed that execute multiple applications in the same MRE instance (i.e., in the same OS process). A multitasking implementation of an MRE can provide better memory usage and faster startup while maintaining portability, mobility, and type-safety.

The multi-tasking MRE we focus on in the second part of this dissertation is Sun Microsystem Labs' Multi-tasking Virtual Machine (MVM) [29]. MVM is a JVM implementation that co-locates execution of multiple programs in a single operating system process. Each program execution is carried out as a *task*. Tasks are used to implement *isolate*s, which are execution containers for arbitrary programs formally defined by the Application Isolation API (Java Specification Request 121) [57] (similar to AppDomains in Microsoft's CLR [63]). Co-locating programs in the same address space simplifies the virtual machine implementation and enables sharing of the runtime representation of programs and dynamically compiled code. Such sharing avoids duplicated effort across programs (e.g. loading, verification) and amortizes runtime costs, such as dynamic compilation, over multiple program instances.

Isolates provide a program with the illusion of a stand-alone JVM. Programs have the same behavior as if they were running on a private JVM. Each isolate has its own primordial loader and hierarchy of class loaders. No sharing of objects can take place between isolates, and the JVM safeguards against inter-isolate interference.

Each task in MVM is associated with a unique task identifier. A task identifier is an index into tables used in MVM to mediate access to data structures that must be replicated on a per-task basis, such as, the task specific part of the runtime representation of a class. All threads running in the context of a given task, are associated

with the identifier of that task as well as other relevant task-specific information. We next describe the MVM features that are pertinent to memory management.

Class Sharing

MVM substantially reduces the footprint of programs by implementing a form of sharing of the runtime representation of classes called *task re-entrance* [32]. Task re-entrance is supported only for classes defined by class loaders, whose behavior is fully controlled by the MVM. This includes the *primordial* and *system* loader of each isolate.

The primordial loader is a special class loader that bootstraps class loading. It is used to load the *base* classes that are intimately associated with a JVM implementation and are essential to its functioning (such as classes of the `java.*` packages). The system loader is the loader that defines the main class of a program. It typically obtains class files from the local file system at a fixed location specified at program start-up.

The system loader serves class loading requests by first delegating them to the primordial loader, and only defines classes that the primordial loader does not define. This behavior is predictable since for a given class path, a class loaded by a primordial or a system loader of any task is always built from the same class file. Further,

symbolic references from classes defined by a primordial or a system loader always resolve identically across tasks.

This allows for a simplified form of sharing where only the task-dependent parts of the runtime representation of a class, such as static variables, class initialization state, protection domain, instance of `java.lang.Class` etc., must be replicated per loader. All other class information, in particular those derived from resolved symbolic links, such as field offsets, virtual table indexes, static method addresses, etc., can be shared across loaders, further increasing the amount of sharing. Access to the task-private part of the representation of a class shared across multiple tasks is mediated via a table indexed by a task identifier (task id). Sharing is not supported for classes defined by program-defined loaders. Instead of a table of task-private class representations, the class representation includes a single task-private representation. Both the interpreter and code produced by the dynamic compiler are aware of this organization and access the task-dependent class information using the task identifier of the current thread.

An extensive description of how MVM implements sharing of the runtime representation of classes, including bytecode and code produced by the dynamic compiler, is described in [27].

Garbage Collection in MVM

The MVM derives from the HotSpot^{TM} Java virtual machine [77]. The current prototype of the MVM [92] retains the heap layout of the original HotSpot JVM and introduces minor changes. Heap management follows a generational strategy based on three generations – permanent, tenured, and young. The permanent generation is a special generation used for allocating objects that constitute the runtime representation of classes and string literals. In the MVM, the permanent generation also includes task tables associated with the runtime representation of task-reentrant classes. Note, however, that we do not allocate the task-private representation of a task re-entrant class, which holds static variables etc., in the permanent generation but, rather, in the tenured generation. The rationale for this is that in the MVM, the lifetime of the sharable part of the runtime representation of a class is much longer. The sharable part's lifetime may range from the lifetimes of a few tasks to the lifetime of the virtual machine itself, unlike the task-private part, which lasts no longer than the duration of the task. The task-private part of the runtime representation of classes is allocated directly in the tenured generation. This avoids cluttering the young generation with objects known to be long lived.

Program threads allocate from the young generation. As in the original implementation of the HotSpot JVM, the young generation is divided into an allocation

space (the *eden*), and a mature space [1], which consists of a pair of equally sized semi-spaces (a *from* and *to* space). Garbage collection of the young generation uses a copying scavenger that evacuates live objects from the *eden* and *from* spaces to the *to* space according to a design similar to that in [93]. Mature objects that have survived several scavenge cycles are promoted to the old generation. Objects from the young generation are never promoted to the permanent generation.

The eden space is used for the vast majority of allocations. Objects that do not fit in the young generation are allocated directly in the tenured generation. To increase per-thread locality and to avoid the cost of atomic instructions in allocation code, the system allocates a thread local allocation buffer (TLAB) from the eden space for threads of tasks. Write barriers for tracking cross-generation pointers follow a card-marking scheme.

The HotSpot JVM supports several algorithms for the tenured generation, but MVM currently only supports mark and compact. Both minor and major collections require bringing all threads to a safepoint in order to proceed. In the case of MVM, all threads of all tasks must be at a safepoint.

The changes introduced by MVM to garbage collection are related to reclaiming space used by terminated tasks. MVM maintains a list of terminated tasks that is purged on a garbage collection. During collection, the list of terminated tasks is

[1] Should not be confused with the old generation

used to scan task tables of the runtime representation of classes, and other task tables referring to heap objects, to zero-out the entries corresponding to terminated tasks, so that no objects of the terminated task are reachable from any live root. This clean up is performed at garbage collection time rather than at task termination, since (i) the heap space used by terminated task cannot be reclaimed without performing a full GC, and (ii) postponing clean up until GC enables the system to factor out the cost of clearing dead references from entries of task tables corresponding to terminated tasks.

The text of chapter 2 is in part a reprint of the material as it appears in the proceedings of the International Symposium on Memory Management (ISMM 2006).

Chapter 3

Application-Specific Garbage Collection

The next generation of high performance server systems must provide continuous availability and high performance to gain widespread use and acceptance. These systems run a single virtual machine (VM) image persistently, and applications and code components can be uploaded and executed as needed by customers (for customization, collaboration, distributed execution, etc.).

Given this model of a single persistent VM, and existing JVM technology, a single, general-purpose collector and allocation policy must be used for all applications. However, many researchers have shown that there is no single combination of a collector and an allocator that enables the best performance for all applications, on all hardware, and given all resource constraints [5, 40, 101].

We therefore investigate whether the choice of the garbage collection policy in MREs should be *application-specific*. To this end, we first present experimental re-

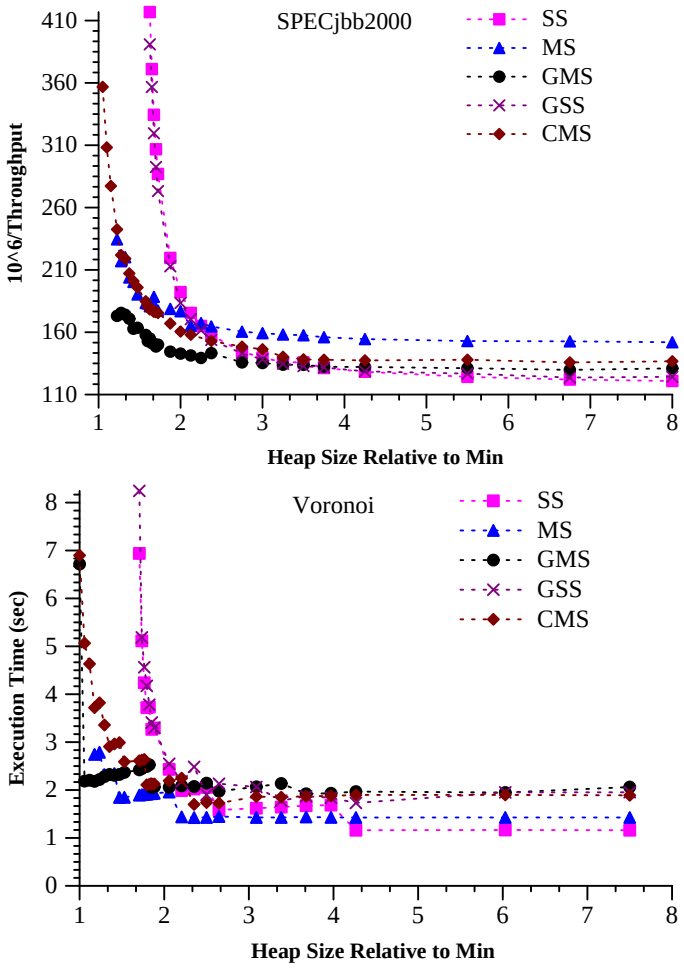


Figure 3.1: Performance using different GCs and heap sizes. The y-axis is total time in seconds. For SPECjbb, we report 10^6/throughput to maintain visual consistency. The x-axis is heap size relative to the minimum with the GC that can execute the program in smallest heap.

sults for benchmark execution time using a wide-range of heap sizes, in Figures 3.1 and 3.2. This set of experiments confirms similar findings of others [5, 40, 101] that indicate that no single GC system enables the best performance for all applications, on all hardware, and given all resource constraints.

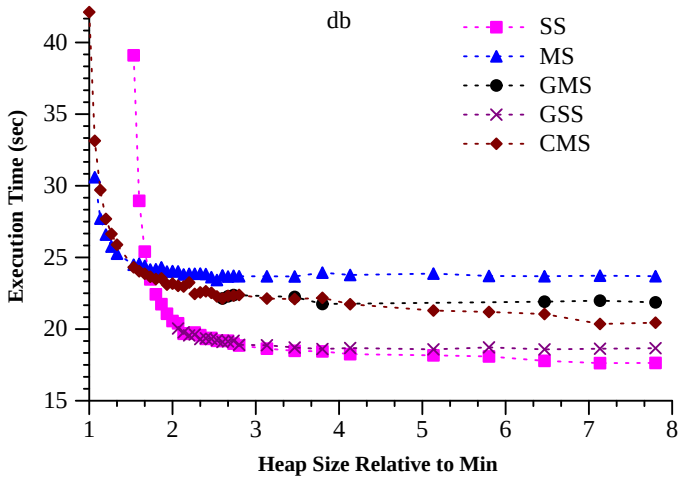


Figure 3.2: Performance using different GCs and heap sizes. The y-axis is total time in seconds.

The graphs show execution time over heap sizes with different garbage collectors for a few standard benchmarks – SPECjbb [84], Voronoi from the JOlden benchmark suite [20], and db from the SpecJVM98 suite [84]. We employ the widely used Jikes Research Virtual Machine (JikesRVM) [2] for our experimentation and prototype system. The x-axis is heap size relative to the minimum heap size that the application requires for complete execution across all GC systems. For SPECjbb, the y-axis is the inverse of the throughput reported by the benchmark; we report 10^6/throughput to maintain visual consistency with the execution time data of the other benchmarks. Lower values are better for all graphs.

The top-most graph in the figure shows that for SPECjbb, the semispace (SS) collector performs best for all heap sizes larger than 4 times the minimum, and the generational/mark-sweep hybrid (GMS) performs best for small heap sizes. The

middle graph, for Voronoi shows that for heap sizes larger than 4 times the minimum, semispace (SS) performs best. For heap sizes between 2 and 4 times the minimum, mark-sweep (MS) performs best. Moreover, for small heap sizes GMS performs best. The bottom-most graph shows the performance of db: SS and GSS (a generational/semispace hybrid) perform best for large heap sizes, while CMS (a non-generational semispace copying/mark-sweep hybrid), and MS perform best for small heap sizes. The collectors will be described in detail shortly. These results support the findings of others [5, 40, 101], that no single collection system enables the best performance across benchmarks. Further, no single system performs best *across heap sizes for a single benchmark/input pair*. We refer to any point at which the best performing GC changes as a *switch point*.

To exploit this execution behavior that is specific to both the application and underlying resource availability, we extended JikesRVM, to enable dynamic switching between GCs. The goal of our work is to enable application-specific garbage collection, to improve performance of applications for which there exist GC switch points, and to do so without imposing significant overhead. Such a system will enable users to extract the best performance from their applications with such an MRE. Moreover, an MRE with GC switching functionality will be able to adapt to enable high-performance for future and emerging applications with little or no change to the MRE.

3.1 Support for Garbage Collection Switching

In this section, we describe various technical issues involved in enabling support for switching between garbage collectors at execution time.

Key to switching between collectors is efficient use of the available virtual address space between GCs. Different GC algorithms expect different heap layouts, such as a mark-sweep space, nursery, or a large object space. Virtual address space is limited and controls the maximum size of the heap. Hence, to make the best use of total available space, the virtual space must be divided carefully between different heap layouts.

In addition to virtual address space considerations, we need to support diverse object header information needed by copying and mark-sweep collectors. Copying and mark-sweep use the object header for different purposes, and support for both techniques involves enabling the use of state information used by both.

Another key concern is *specialization*. For performance, code is specialized for the current garbage collector. For instance, inlining of allocation sites, and the presence of write barriers based on whether or not the current garbage collector is generational. Since the garbage collector may change at runtime, assumptions made during compilation for specialization may change as well. Consequently, we must be able to invalidate (recompile) specialized methods, and in addition, replace specialized

code that is *executing* at the time of the switch. We shall describe mechanisms for invalidation in Section 3.2.

We discuss each of the above issues in detail below.

3.1.1 Multiple Garbage Collectors in a Single JVM

JikesRVM [2] is an open-source virtual machine for Java that employs dynamic and adaptive optimization with the goal of enabling high performance in server systems. JikesRVM compiles Java bytecode programs at the method-level at runtime (just-in-time), to x86 (or Power PC) code. JikesRVM supports extensive runtime services – garbage collection, thread scheduling, synchronization, etc. In addition, JikesRVM implements adaptive or mixed-mode optimization by performing on-line instrumentation and profile collection, and then uses the profile data to evaluate when program characteristics have changed enough to warrant method-level re-optimization. The current version of the JikesRVM optimizing compiler applies three levels of optimization (0, 1 and 2). Level 0 optimizations include local propagation (of constants, types, copies), arithmetic simplification, and check elimination (of nulls, casts, array bounds). Moreover, as part of level 0 optimizations, write barriers are inlined into methods if the GC is generational. Level 1 optimizations include all of the level 0 optimizations as well as common sub expression elimination, redundant load elimination, global propagation, scalar replacement, and method inlining (including calls

to the memory allocation routines). Level 2 includes SSA based optimizations in addition to level 1 optimizations.

JikesRVM (version 2.2.0+) uses the Java Memory Management Toolkit (JMTk) [12] that enables garbage collection and allocation algorithms to be written and "plugged" into JikesRVM. The framework offers a high-level, uniform interface to JikesRVM that is implemented by all memory management routines. A GC is a combination of an allocation policy and a collection technique (this corresponds to a *Plan* in JMTk terminology). The JMTk provides the functionality that allows users to implement their own GC (without having to write one from scratch) and to perform an empirical comparison with other existing collectors and allocators. For this purpose, it provides users with utility routines for common GC operations, such as, copying, marking and sweeping objects. When a user builds a configuration of JikesRVM, she is able to select a particular GC for incorporation into the JikesRVM image.

The five GCs that we consider in this work are Semispace copying (SS), a Generational/Semispace Hybrid (GSS), a Generational/Mark-sweep Hybrid (GMS), a non-generational Semispace/ Mark-sweep Hybrid (CMS), and Mark-sweep (MS). These systems use stop-the-world collection and hence, require that all mutators be paused when garbage collection is in progress. Semispace copying and mark-sweep are standard non-generational collectors [61, 12] with a single space for most mutator allocation (large objects are allocated in a separate space). Allocation in the semispace con-

figuration is through a pointer increment (bump pointer), while that in mark-sweep involves a segregated free list. The free list is divided into several size classes and objects are allocated from the appropriate size class using a first-fit algorithm. Non-generational collectors collect the entire heap on every collection. Bump pointer allocation is believed to be much faster than free list allocation, since it is a much simpler operation.

The generational collectors, GSS and GMS, make use of well-known generational garbage collection techniques [3, 95]. Young objects are allocated in an Appel-style [3] variable-sized nursery space using *bump pointer* (pointer increment) allocation from a contiguous block of memory. The boundary between the nursery and the mature space is dynamic. Initially, the nursery occupies half the heap, and the mature space is empty. As live data from the nursery is promoted to the mature space on a minor collection, the size of the nursery shrinks. After a major (full heap) collection, the mature space contains live old data, and the nursery occupies half of the remaining space. Upon a minor collection, the nursery is collected, and the survivors are copied (promoted) to the mature space. Promotion is *en masse*, i.e., all survivors are copied to the mature space without first being moved to an intermediate space [91]. The mature space is collected by performing a full heap collection. This process is referred to as a *major collection*. Since minor collections are performed separately from major collections, pointers from mature space objects to nursery objects must

be identified to keep the corresponding nursery objects live. A *write barrier* is employed for this purpose. A write barrier is a series of instructions that is used to keep track of such mature space objects.

The main difference between GSS and GMS is the way in which the mature space is collected. GSS employs copying collection for this purpose, while GMS makes use of mark-sweep collection.

During a minor collection, nursery objects can be copied to the mature space in the GSS collector by a simple bump pointer allocation. However, allocation from the mature space in GMS is performed using a sequential, first-fit, free-list. GMS mature space collection is a two-phase process that consists of a mark phase in which live objects are marked, and a sweep phase in which unmarked space is returned to the free-list.

Generational GC performs well when the number of minor collections is large, since a minor collection is much faster than a full heap GC. However, when memory is plentiful, and GC is not required, non-generational collection may perform competitively. In fact, under these conditions, in several cases, semispace collection may commonly outperform other GCs due to cache locality benefits, and low fragmentation enabled by bump pointer allocation [61].

CMS [1] is similar to a traditional semispace collector (SS) in that it is non-generational and is divided into two memory regions. However, CMS is a hybrid approach in which the first section is managed via bump pointer allocation and copy collection and the second section is managed via Mark-sweep collection (and uses free-list allocation as described above). CMS does not use write barriers. As a result, CMS is only able to identify references from the mark-sweep space to the semispace by tracing the objects in the former. Consequently, when a CMS collection occurs, the entire heap is collected – using copy collection for the first section then mark-sweep collection for the second section. CMS is more space efficient compared to a copying collector since it does not require a copy reserve. It is supposed to achieve good performance when the number of objects promoted is low.

JVM system classes and large objects are handled specially in JikesRVM/ JMTk system. There is a separate immortal space that holds the JikesRVM system classes. Allocation in the immortal space is via the bump pointer technique and this space is never collected. In addition, objects of size 16KB and greater are considered as "large objects". The large object space is managed using mark-sweep collection. Collectors that employ a mark-sweep space, allocate large objects from the mature space (since these collectors already employ mark-sweep collection), while copying collectors employ a separate large object space.

[1] This is a stop-the-world collector and should not be confused with the Concurrent Mark-Sweep collector in Sun's HotSpot VM [91].

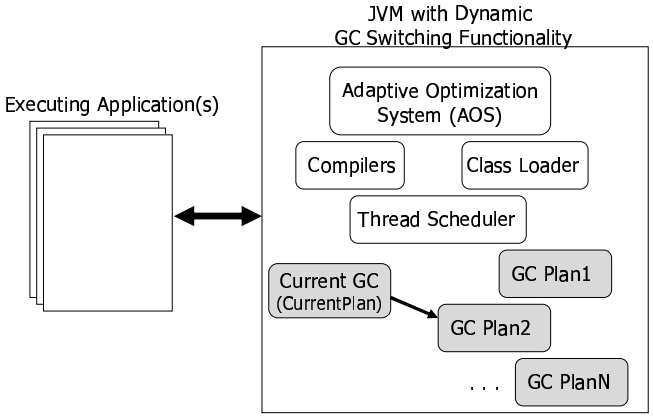


Figure 3.3: Overview of our GC switching system. The JVM consists of the standard set of services as well as *multiple* GCs (an allocator and collector) as opposed to one per JVM image. The system employs the current GC through a reference to it called the CurrentPlan. When a switch occurs, the system updates the CurrentPlan pointer (and performs a collection in some cases). All future allocations and collections (if any) use the newly selected GC.

Figure 3.3 shows the design of our GC switching system. Each MRE image contains multiple GCs in addition to a set of standard services, such as the class loader, compilers, optimization system, and thread scheduler. Each GC consists of an implementation of an allocator and a collector.

The system switches to a new GC when doing so will improve performance. The system considers program annotations (if available), application behavior, and resource availability to decide when to switch dynamically, and to which GC it should switch. The GC currently in use is referred to by a CurrentPlan pointer. The compiler and runtime use this pointer to identify and employ the currently available GC. When a switch occurs, the system updates CurrentPlan to point to the new GC and

performs allocation and collection (if needed) using the newly selected allocation and collection algorithms.

Each GC in JikesRVM/JMTk is implemented as a *Plan*. The plan identifies the type of allocator and collector that is built into the image and consists of a set of classes that implement the appropriate algorithms for collection (semispace, generational, etc.) and allocation (free-list, bump pointer, etc.).

We extended JikesRVM/JMTk system to implement each of JikesRVM GCs within a single JikesRVM image. We show the original and new JikesRVM/
JMTk class hierarchy in Figure 3.4. We implemented these classes so that much of the code base is reused across collection systems. The size of the MRE image built with our extensions is 44.2MB (with the boot image compiled using the optimizing compiler at level 1), compared to an average size of 42.6MB for the reference JikesRVM images (ranging from 37.2MB for SS to 49.4MB for MS) – our extensions do not significantly increase code size. Interestingly, the reference JikesRVM image when built with MS, is larger than an image built with our modifications. We believe that the reason for this is that inlining of allocation sites for mark sweep increases code size significantly. Note that we do not inline allocation sites for boot image code in case of our switching system.

To support multiple GCs, we require address ranges for all possible virtual memory resources to be reserved. Our goal is to enable maximum overlap of virtual ad-

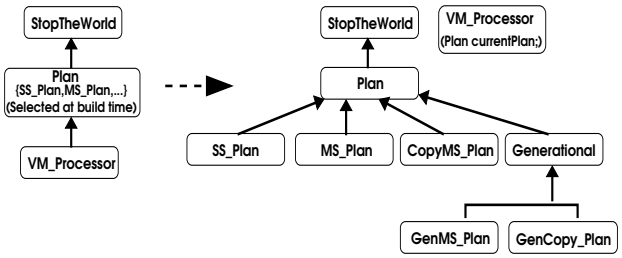


Figure 3.4: JikesRVM/JMTk class hierarchy: Original and switch-enabled.

dress spaces to reduce the overhead of switching. Our address space layout is shown in Figure 3.5(a). Each address range is lazily mapped to physical memory (as it is used by the executing program), in 1 Megabyte chunks. There are three shared spaces that we inherit from the default JikesRVM implementation: the immortal (uncollected), GC Data Structure area (uncollected), and large object (>16KB) space. The GC that is currently in use employs a subset of other spaces as appropriate.

Switching Between GCs

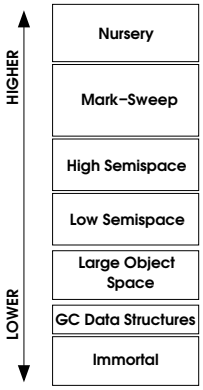
Switching between GCs requires that all mutators be suspended to preserve consistency of the virtual address space. Since the JikesRVM collectors are stop-the-world, JikesRVM already implements the necessary functionality to pause and resume mutator threads. We extended this mechanism to implement switching.

During a GC switch operation, we stop each executing mutator thread as if a garbage collection were taking place. A full heap GC, however, may not be necessary for all switches. To enable this, we carefully designed the layout of our heap spaces

(Figure 3.5(a)) in such a way as to reduce the overhead of collection, i.e., to avoid a full garbage collection for as many different switches as possible. For example, a switch from SS to GSS only requires that future allocations by the application use the GSS nursery area since SS and GSS share two half-spaces. Therefore, we only need to perform general bookkeeping to update the *CurrentPlan* to implement the switch.

Figure 3.5(b) indicates whether a GC is required, for a switch from the row GC to the column GC, and if it is, the type of GC required, e.g., full (F), minor (M), or none (N). We use the notation XX→YY to indicate a switch from collection system XX to collection system YY. The entries in the table show the type of GC that is required for row→column. Note that we need to perform a garbage collection when switching from MS in only two cases (while switching to SS and GSS, the latter being a collector that is very often not the best choice, hence is not a frequent scenario). Moreover, MS commonly works well for very small heap sizes. We therefore use MS as our initial, default collector. As our system discovers when to switch to a more appropriate collection system, the cost of the switch itself is likely to be low.

We next describe the operations required for each type of switch. Whenever we perform a copy from one virtual memory resource to another, we use the allocation routine of the GC to which we are switching.



(a)

GC Requirements Upon Switch

N:None, F:Full, M:Minor

Switch from Row to Column

	SS	CMS	GMS	GSS	MS
SS	–	F	F	–	F
CMS	F	–	F	F	F
GMS	F	–	–	F	M
GSS	M	F	F	–	F
MS	F	–	–	F	–

(b)

Figure 3.5: Virtual address space layout in the switching system (a) and a table (b) that indicates when a GC is required on a switch (from the row GC to the column GC) and its type: full (F), minor (M), or none (–).

Switches That Do Not Require Collection. As mentioned above, SS→GSS, MS→CMS, MS→GMS, and CMS→GMS do not require a collection since their virtual semispaces are shared.

Switches That Require Minor Collection. When we switch from a generational to a similar non-generational collector, e.g., GMS→MS and GSS→SS, we need only perform a minor collection. That is, in addition to updating the *CurrentPlan*, we must collect the nursery space and copy the remaining live objects into the (shared) mature space.

Switches That Require Full Collection. The remaining switch combinations require a full garbage collection. We perform each switch as follows:

- **SS/GSS→GMS/CMS/MS**. To switch between these collection systems, we perform a semispace collection (or a major collection for GSS). However, instead of copying survivors to the empty semispace, we copy them to the mark-sweep space of the target systems. When switching from GSS, we do the same; however, we must also copy the objects in the GSS mature space to the mark-sweep space.

  Collectors that use semispaces (SS and GSS), require a copy reserve area, and consequently, do not perform well under memory pressure. In addition, if the ratio of live objects to dead is high, copying collectors involve expensive copying of live objects. Under such conditions, it would be beneficial to switch to a non-copying GC.

- **GMS/MS→SS/GSS**. To perform this switch, we perform a major collection and copy survivors from the nursery and live objects from the mature space to the semispace. If we are switching from a non-generational MS system to SS or GSS, we mark live objects in the mark-sweep space and we forward them to the semispace resource. Since we must move objects during MS collection, we must maintain multiple states per object. We do this using an efficient, multi-purpose, object header described in Section 3.1.2.

If memory is plentiful, copying collectors can provide good performance since they employ fast, bump-pointer allocation. Also, certain applications might fragment the heap excessively, requiring compaction, which is inherently provided by copying collectors. Copying collection, is also supposed to provide better data cache locality, since objects are laid out in allocation order.

- **CMS→Any GC**. Since there are no write barriers implemented for CMS, the heap spaces in this hybrid collector cannot be collected separately. Without write barriers to identify references from the mark-sweep space to the semispace, we may incorrectly collect live objects if we collect the semispace alone, i.e., those that are referenced by mark-sweep objects but not reachable from the root set. When we switch from CMS to any other GC, we must perform a full collection to ensure that we consider all live objects.

  CMS is a compromise between generational, and non-generational collection. It does not incur the penalty of a write-barrier during mutation, yet provides segregation of old objects from young. However, CMS does not provide incremental behavior, i.e. the ability to collect only a part of the heap (usually, the one with most likelihood of dead objects), independently of other parts, that generational collectors achieve.

Although the switching process is specific to the old and the new GCs, we provide an extensible framework that facilitates easy implementation of switching from any GC to any other, existing or future that is supported by JikesRVM JMTk. Moreover, unlike prior work, our system is able to switch dynamically between GCs that use very different allocation and collection strategies.

When a switch completes, we suspend the collector threads and resume the mutators, as is done during the post-processing of a normal collection. In addition, we *unmap* any memory regions that are no longer in use.

A limitation of the switching mechanisms described above is that we may not be able to perform certain kinds of switches when memory is highly constrained. For example, while switching from MS (or GMS, CMS) to SS (or GSS), we need to map the virtual address space corresponding to the SS *tospace*, on demand. However, we cannot unmap the MS address space until all live objects have been copied to the SS tospace. Consequently, our system requires more mapped memory than the reference system, *while* performing the switch in these cases. In practice however, switching from MS to SS or GSS when memory is constrained would be a poor choice (we provide further explanation of why this is the case in Section 3.3). A similar problem exists for switching from SS (or GSS) to a MS (or GMS, CMS) system. Note, however, that in these cases, we can unmap memory from the SS

tospace before we copy objects to the MS space, since the SS tospace will not be used subsequently.

3.1.2 Multi-purpose Object Header

As mentioned in the previous section, to switch from a GC that uses a mark-sweep space (GMS, CMS, and MS) to a GC that uses a contiguous semispace (GSS, SS), we must maintain state for the mark-sweep process as well as for each object's forwarded location that is used by copying collection. Typically, garbage collectors store this state in the header of each object. In JikesRVM, the garbage collectors each use a single 4-byte entry in the object header, called the *status word*.

The mark-sweep collector requires two bits in the status word: the *mark bit* to mark live objects and the *small object bit* to indicate that the object is a small object. The use of the *small object bit* enables efficient size-specific free-list allocation. Since the system aligns memory allocation requests on a 4-byte boundary, the lowest two bits in an object's address are always 0. Hence, the *mark bit* and the *small object bit* can be encoded as the lowest two bits in the status word.

Semispace collectors also require header space to record the state of the copy process and the address to which the object is copied. A semispace collector marks an object as *being forwarded* while it is being copied. Once it is copied, the object is marked as *forwarded* and a forwarding pointer to the location to which the object

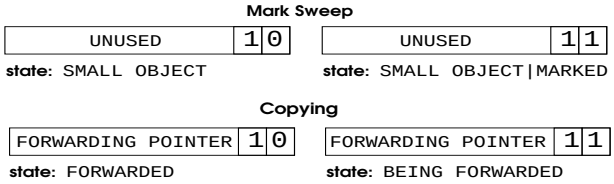


Figure 3.6: Examples of bit positions in status word in object header

was copied, is stored in the initial 30 bits of the header. The *being forwarded* state is necessary to ensure synchronization between multiple collector threads. These two states are stored in the two least significant bits of the status word.

The two least significant bits in an object status word implement different states depending on the collector. For example, as shown in Figure 3.6, if JikesRVM is built using a mark-sweep GC, the value `0x2` in the two least significant bits of the status word of an object indicates that the object is small and unmarked. However, if instead, the semispace collector is used, this state indicates that the object has been forwarded to the to-space during a collection. Similarly, if both bits are set, the status word indicates that the object is a small object and has been marked as live by a mark-sweep collector; the same state indicates to a semispace collector thread that the object is currently being forwarded by another thread.

Upon a switch from a collector that uses a mark-sweep space to one that uses a semispace, we must forward marked objects to the semispace. Consequently, our switching system must support *all four distinct states*, in addition to space for a forwarding pointer. To account for the two additional bits required and to avoid using an

additional 4-byte header entry, we use bit-stealing (also used in prior GC systems [8]) in which we "steal" the two least significant bits from another address value that is byte-aligned.

The object header in JikesRVM also stores a pointer to a Type Information Block (TIB) data structure, which provides access to the internal class representation and the virtual method table of the object. We use the two least significant bits from the TIB pointer to store the additional states, *being forwarded* and *forwarded*, during the copying process. This implementation requires that we modify VM accesses to the TIB so that these bits are ignored. We found that this does not introduce significant overhead.

3.2 Specialization Support for GC Switching

A naïve switching implementation would involve two primary sources of overhead: write barriers are not needed by all collectors, and the loss of inlining opportunities due to dynamically changing allocation routines. Since our system can switch to a generational collector at any time, we would need to insert write barriers for every pointer field assignment in every method – these instructions would execute even when the collector in use is non-generational. Moreover, if the GC does not change over the lifetime of the program, we can inline calls to the allocation routine.

However, in our system, the allocation routine may change, precluding our ability to inline.

To avoid a loss in performance due to these two issues, we *specialize the code* for the underlying GC aggressively and speculatively. That is, we inline allocation routines and insert write barriers only if the underlying GC is a generational collector.

For these specializations, we consider only optimized code. JikesRVM, like many other commonly used JVMs [91, 24, 77], employs adaptive optimization in which it only optimizes code that it identifies as hot, using efficient, online sampling of the executing program. JikesRVM optimizing compiler applies three levels of optimization (0, 1, and 2) depending on how "hot" a method is. Level 0 optimizations include local propagation (of constants, types, copies), arithmetic simplification, and check elimination (of nulls, casts, array bounds). In addition, this level includes the inlining of write barriers into methods if the GC is generational. Level 1 optimizations include all level 0 optimizations as well as common sub expression elimination, redundant load elimination, global propagation, scalar replacement, and method inlining (including calls to allocation routines). Level 2 optimizations include all of level 1 optimization plus SSA-based transformations.

All unoptimized methods are compiled by JikesRVM using a fast compiler that applies no optimization. We modified this compiler to insert write barriers into all methods regardless of the underlying collector. Since JikesRVM itself is written in

Java, all MRE methods are compiled into a boot image – we modified this process as well to insert write barriers and to avoid inlining allocation routines into boot image methods. To enable speculative specialization, we modified level 0 of the optimizer so that it checked the CurrentPlan to determine whether to insert write barriers. We also modified level 1 (and above) to inline the allocation routines of the CurrentPlan collector. We made these changes in the runtime compiler (as opposed to the boot image compiler),

For annotation-guided GC selection, our system switches GCs immediately prior to the start of program execution. Therefore, no methods have been optimized. Moreover, once the program begins, the system does not perform switching again. Thus, our specialization for write barriers and allocation routines is always correct in this case.

However, for automatic switching, the system can (and does) switch at any time. We therefore require a mechanism to "undo" the specializations when a switch occurs. We need only undo specializations that will cause incorrect execution. There are two such cases. First, the prior GC was not generational, the new GC is generational, and there is a field update in an optimized method. The new GC, therefore, requires a write-barrier for correctness. Second, there is an allocation site in an optimized method and the optimization level used by the compiler was 1 or higher. Consequently, the existing inlined allocation sequence is no longer valid

and must be invalidated. For future invocations of these methods, we use method invalidation [48] to undo the specialization. For methods that are currently executing, i.e., those that are on the runtime stack, we require on-stack-replacement (OSR) [48, 22, 47, 38, 77, 91, 88, 46] of the method.

To enable OSR, the compiler must track the program execution state of the method at a particular program point in native code. The execution state consists of values for bytecode-level local variables, stack variables, and the current program counter. The execution state is a map that provides the OSR system with runtime values at the bytecode-level (source-level) so that the system can recompile and restart the method using another version. Existing OSR implementations insert a special (pseudo-) instruction, called an OSR point, to enable state collection.

OSR for replacement of executing optimized methods (as is needed for specialized methods in the GC switching system) is more complex than for unoptimized methods since compiler optimization can eliminate variables, combine multiple variables into one, and add variables (temporaries). This makes the ability to map bytecode-level variables correctly very challenging. All extant approaches to OSR avoid optimization across OSR points to avoid adding complexity to the compilation system. This, however, as we will later show, can significantly degrade code quality (and thus performance) if OSR support is to be enabled at a significant number of program points.

A Novel OSR Implementation

There are two reasons why extant approaches to OSR can degrade performance. First, all method variables (locals as well as stack) are considered live at an OSR point; by doing so, the compiler artificially extends the live ranges of variables and significantly limits the applicability of optimizations such as dead code elimination, load/store elimination, alias analysis, and copy/constant propagation. Second, OSR points are "pinned" in the code to ensure that variable definitions are not moved around the OSR points; this precludes optimization and code motion across OSR points.

These prior implementations do not negatively impact performance (as a result of poor code quality) significantly when there are only a small number of OSR points. However, our switching system requires an OSR point at every point in the code at which a switch can occur; these are the points at which a GC can occur, i.e., gc-safe points. GC-safe points in JikesRVM include implicit yield points (method prologues, method epilogues, and loop back-edges), call sites, exception throws, and explicit yieldpoints.

Since our GC switching system requires a very large number of OSR points, many along the critical path of the program, existing OSR implementations can severely degrade the performance of our GC switching system. We therefore extended JikesRVM OSR implementation with a novel extension that is more amenable

to optimization. In particular, we automatically track compiler optimizations in a specialized data structure to hold state information, called a variable map (VARMAP).

A VARMAP is a per-method list of bytecode variables (primitives as well as reference types) that are live at each gc-safe point. This list is independent of the code and does not impact the liveness information of the program point, nor does it restrict code motion optimizations. To ensure that we maintain accurate information in the VARMAP, we update it incrementally as compiler optimizations are performed. The VARMAP is somewhat similar in form to the data structure described in [36], which was used to track pointer updates in the presence of compiler optimizations, for garbage collection support in Modula-3. However, unlike prior work, we track all stack, local, and temporary variables online, across a wide range of compiler optimizations automatically and transparently, during just-in-time compilation and dynamic optimization of Java programs.

Figure 3.7 shows an example of a VARMAP entry for a snippet of Java source. We include the equivalent Java bytecode and JikesRVM high-level intermediate representation (HIR) of the code. Below the code, we show the VARMAP entry for the `callme()` call site. which contains the next bytecode index (25) after the call site `callme` and three typed local variables (`a: l8i, b: l15i, c: l17i`).

To update the VARMAP entries, we defined the following system methods:

Source	Byte Code	VARMAP entry

..
int c,d;
b = a;
c = b * 4;
callme();
d = a + b;
..
	..
	14: iload_1
	15: istore_2
	16: iload_2
	17: iconst_4
	18: imul
	19: istore_3
	20: invokestatic #3 //callme()V
	23: iload_1
	24: iload_2
	25: iadd
	26: istore_4
	..
		..
		15: int_move l15i(int) = l8i(int)
		18: int_shl l17i(int) = l15i(int), 2
		20: call static "callme() V"
		25: int_add l19i(int) = l8i(int),
		l15i(int)
		..
		Intermediate Code (HIR)
		25@main (..LLL,..),.., l18i(int), l15i(int), l17i(int), ..
		bcindex: 25, L: local var, a: l8i, b: l15i, c: l17i

Figure 3.7: Shows how the VARMAP is maintained for a snippet of Java source (its bytecode and high-level intermediate representation (HIR) is included). We show the VARMAP entry for the `callme()` call site that contains the next bytecode index (25) after the call site `callme` and three local variables with types (`a: l8i, b: l15i, c: l17i`).

Intermediate Code (HIR)	.. 15: int_move l15i(int)=l8i(int) 18: int_shl l17i(int)=l15i(int), 2 20: call static "callme() V" 25: int_add l19i(int)= l8i(int), l15i(int) ..	.. 15: int_move l15i(int)=l8i(int) 18: int_shl l17i(int)=**l8i(int)**, 2 20: call static "callme() V" 25: int_add l19i(int) = l8i(int), **l8i(int)** ..
VARMAP entry	25@main (..LLL,..),.., l18i(int), l15i(int), l17i(int), ..	25@main (..LLL,..),.., l18i(int), **l8i(int)**, l17i(int), .. **transferVarForOsr(l15i, l8i)**
	Before optimization	**After optimization**

Figure 3.8: Shows how the VARMAP is updated after copy propagation. Variable `b: l15i` is replaced with `a: l8i`.

- *transferVarForOsr(var1, var2)*: Record that `var2` will be used in place of `var1` from here on in the code (e.g., as a result of copy propagation)

- *removeVariableForOsr (var)*: Record that `var` is no longer live/valid in the code. Note that, even though a variable may not be live, we must still remember its relative order among the set of method variables.

- *replaceVarWithExpression(var, vars[], operators[])*: Record that variable `var` has been replaced by an expression that is derivable from the set of variables `vars` and `operators`.

Our OSR-enabled compilation system handles all extant JikesRVM optimizations at all optimization levels. Each time a variable is updated by the compiler, the update occurs through a wrapper function that automatically invokes the necessary VARMAP functions. This enables us to easily extend the compilation system with new optimizations that automatically update the VARMAP appropriately. For example, for copy and constant propagation and CSE (common sub-expression elimination), when a use of a variable is replaced by another variable (or constant), the wrapper function performs the replacement in the VARMAP record by invoking the `transferVarForOsr` function as shown in Figure 3.8 for an update that results from copy propagation.

We also update the VARMAP during live variable analysis. We record variables that are no longer live at each potential OSR point, and record the relative position of each in the map. We set every variable that live-analysis discovers as dead, to a `void type` in the VARMAP. We identify local and stack variables by their relative positions in the Java bytecode. Maintaining the relative positions of variables in the VARMAP allows us to restore a variable's runtime value to the correct variable location.

During register allocation, we update the VARMAP with the actual register and spill locations for the variables, so that they can be restored from these locations during on-stack replacement. The VARMAP contains symbolic registers corresponding to each variable. We update symbolic registers with a physical register or a stack location upon allocation by querying the map maintained by the register allocator for every symbolic register that has been allocated to a physical register. We record spilled variables via the spill location that the allocator encodes as a field in the symbolic register object.

When the compilation of a method completes, we encode the VARMAP of the method using the compact encoding implemented for OSR points in the original system [38]. The encoded map contains an entry for each potential OSR point. Each entry consists of the `register map`, which is a bit map that indicates which physical registers contain references (which a copying garbage collector may update). In

addition, the map contains the current program counter (bytecode index), and a list of pairs `(local variable, location)` (each pair encoded as two integers), for every inlined method (in case of an inlined call sequence). The encoded map remains in the system throughout the lifetime of the program and all other data structures required for OSR-aware compilation (including the original VARMAP) are reclaimed during GC.

Triggering On-Stack Replacement

During execution, following a GC switch, we trigger OSR lazily as is done in Self for debugging optimized code [47]. We tag a specialized method at compile time, and read this tag during GC switch to identify the method as specialized. We modify the return address of the specialized method's callee so that it will jump to a special utility method that performs OSR for the specialized method. By triggering OSR lazily, we eliminate the need for runtime checks in the application code.

The utility method extracts the execution state from the stack frame of the specialized method, and sets up the new stack frame. To preserve register values contained in registers for the execution of specialized methods, the helper saves all registers (volatiles and non-volatiles) into its stack frame. Since the helper is not directly called from the specialized code, we must "fake" a call to the helper. This involves setting the return address of the helper to point to the current instruction pointer in the

specialized code upon entry to the helper. This process also requires that we update the stack pointer for the helper appropriately.

In the next section, we describe two uses of the framework for garbage collection switching – *annotation-guided switching*, and *automatic switching*.

3.3 Annotation-Based Garbage Collector Selection

By implementing the functionality to switch between collection systems while JikesRVM is executing, we can now select the "best performing" collection system for each application that executes using our system. To this end, we implemented *Annotation-guided GC System Selection*. In particular, we use a class file annotation to identify per-application garbage collectors that should be employed by our GC Switching system. We compactly encode the annotation in a class file that contains a `main(...)V` method using a technique that we developed in prior work [66].

To identify the GC that we recommend as an annotation, we analyzed application performance offline using the different JikesRVM GCs. We considered a number of different heap sizes and program inputs. We list the inputs in Figure 3.11 and refer to them as Input and Cross. We extracted, for each heap size, the best performing GC across inputs. In addition, for benchmarks for which there were multiple best

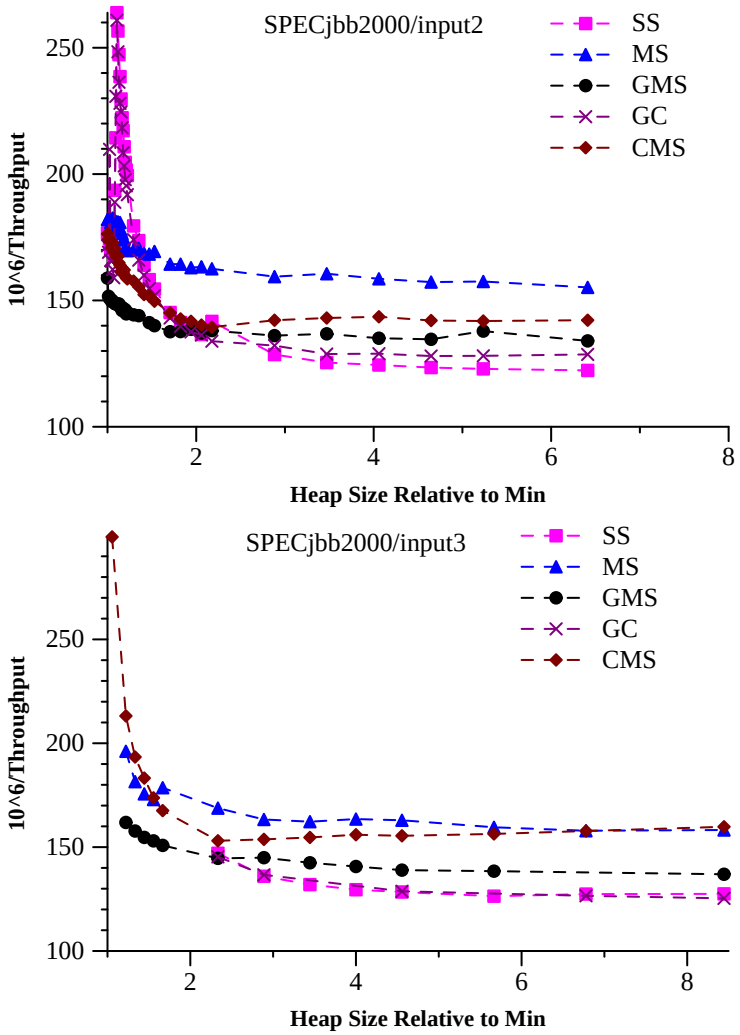


Figure 3.9: Additional inputs for SPECjbb2000 (in addition to input1 in Figure 3.1).

performing GCs for different heap sizes, we also identified the *switch points* for each program, i.e., the heap sizes at which the best performing GC changes.

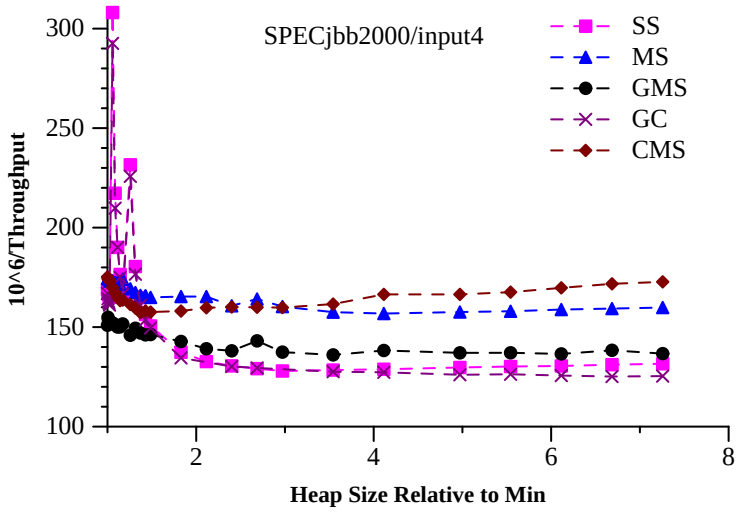


Figure 3.10: Additional input for SPECjbb2000 (in addition to input1 in Figure 3.1).

For all benchmarks that we studied, the per-GC performance was very similar across inputs. Only one benchmark exhibited differences in the best performing GC across inputs (JavaGrande). All other benchmarks showed no change in the choice of the GC across the inputs that we used. To investigate this further, we looked at several inputs for the SPECjbb benchmark, which is an example of a GC-intensive server program. For 4 different inputs for SPECjbb, we found that GMS enables best performance for small or medium heaps, while SS works best for large heaps (see Figures 3.9 and 3.10). This *input independence* appears to be very different from other types of profiles, such as, method invocation counts, field accesses, etc., in which cross-input behavior can vary widely [65, 66]. Therefore, we believe that

Benchmark	Input/Cross	Min Heap (MB)	Annot GC Selector	
			GC(s)	Switch Ratio
compress	100/10	21	SS	—
jess	100/10	9	GMS	—
db	100/10	15	CMS/SS	1.73
javac	100/10	30	GMS	—
mtrt	100/10	16	GMS	—
jack	100/10	18	GMS	—
JavaGrande	AllSizeA/SizeB	15	GMS/SS	3.00
MST	1050 nodes/640	78	MS/CMS	1.47
SPECjbb2000	1 warehouse/2	40	GMS/SS	3.00
Voronoi	65000 pts/20000	34	MS/SS	4.26

Figure 3.11: Inputs that we considered to evaluate GC behavior across heap sizes, the minimum heap size in which the program will run using our JVM, and the GC selection decisions with which we annotate each program to enable annotation-guided GC switching.

it is less likely that we will negatively impact performance for inputs that we have not profiled. To select the GC to provide as an annotation for JavaGrande, we identified the GC that imposed the smallest percent degradation over the best performing collector across inputs at a range of heap sizes.

The values that we annotate are shown in the final two columns of Figure 3.11. For each benchmark, we specify the GC that performs best. If there is more than one best performing GC for different heap sizes, i.e., there is a *switch point*, we annotate each of the GCs and the switch point.

We found that for all of the benchmarks studied, if there was a switch point, there was only a single switch point and that the switch point heap size was very similar relative to the minimum heap size for each input. As such, we specify the switch point as the *ratio* of switch point heap size and the minimum heap size.

At program load time, the JVM computes the ratio, $\frac{current_max_heap_size}{min_heap_size}$, and compares this value with the annotated ratio. If the computed ratio is less, the JVM switches to the first GC, or to the second GC, otherwise. This requires that we also annotate the minimum heap size for the program and input. By doing so, we reduce the amount of offline profiling required by users of our system since, given the minimum heap size for an input, we can compute the switch point using the ratio from any input. We found that the switch point ratio holds across inputs for all of the benchmarks that we studied. Five of the eleven programs have switch points.

3.4 Automatic Garbage Collector Switching

In addition to annotation-guided GC, we investigated a mechanism to guide switching decisions automatically, when resources are suddenly constrained. In this scenario, the operating system (OS) reclaims virtual memory from our JVM for allocation and use by another process. Such a scenario is common to server systems that execute many competing tasks concurrently.

The scenario that we investigated was one in which the program executes using a sufficiently large heap size, e.g., 200MB. During execution the OS reclaims memory and thereby reduces memory that is available for the heap in use by the executing program. In some cases, this may cause an OutOfMemory error when there is not sufficient virtual memory for the program to make progress.

Our switching system has an advantage in these cases since it can switch to a GC that makes more efficient use of the heap when resources are constrained, e.g., a non-copying system vs. a copying collector. We can switch to such a GC and allow the program to make progress and to avoid termination via the OutOfMemory error. In addition, by switching to a system that performs better under restricted resources, we can reduce the number of garbage collections that are performed, which may improve performance.

We employ a set of heuristics to determine when to switch. The GC switching system monitors the time spent in GC versus the time spent in the application threads. When this *GC load* exceeds 1 for an extended period of time, the system switches to a GC that is more appropriate when resources are constrained. In addition, we also switch GCs when we find that garbage collections are being triggered too frequently, measured as the duration for which the application threads execute between successive garbage collections. We initially use semispace copying collection (SS) when the program starts (as opposed to MS for the annotation-driven case). SS is the best

performing collector across the programs that we studied – when the heap size is large (>200MB). When the switch occurs, the system employs Generational/Mark-Sweep (GMS); GMS performs best when resources are constrained. GMS has more available mature space (since it is mark-sweep collected) compared to other generational collectors. GMS performs no copying for the mature space and thus, when the GCs are frequent, less overhead is imposed on the program, unlike a copying collector.

We evaluate these heuristics and scenarios in our evaluation section. Though this set of heuristics is simple, we show that the GC switching functionality can achieve significant performance benefits (as well as avoid OutOfMemory errors). We plan to investigate other opportunities for automatic GC switching, e.g., to improve locality given changes in program phase behavior, as part of future work.

3.5 Evaluation

To empirically evaluate the effectiveness of switching between garbage collectors dynamically, we performed a series of experiments using our system and a number of benchmark programs. We first describe these benchmarks and our experimental methodology with which we generated the results.

3.5.1 Experimental Methodology

We gathered our results using a dedicated 2.4GHz x86-based Xeon machine (with hyperthreading enabled) running Debian Linux v2.4.18. We implemented our switching framework within JikesRVM version 2.2.0 with IBM jlibraries (Java libraries) R-2002-11-21-19-57-19. We employ a pseudo-adaptive JikesRVM configuration [79] in which we capture the methods that JikesRVM identifies as hot in an offline, profiled run. We then optimize those methods when they are first invoked to avoid the JikesRVM learning time [65], to reduce the non-determinism inherent in the adaptive configuration, and to enable the repeatability of our results. The boot image is compiled using the optimizing compiler (level 1).

We measured the impact of switching on application performance separately from compilation overhead. To enable the former, we executed the benchmarks through a harness program. The harness repeatedly executes the programs; the first run includes program compilation and later runs do not since all methods have been compiled following the initial invocation. We report results as the average of the wall clock time of the final 5 of 10 runs through the harness. We experimented with a range of programs from various benchmark suites, e.g., SpecJVM98 and SPECjbb [84], JOlden [20], and JavaGrande [58] – we omit mpegaudio from the SpecJVM suite, since it exhibits very little allocation behavior and does not exercise memory extensively.

3.5.2 Results

We next present the empirical evaluation of our system. We first evaluate the impact of our new, VARMAP-based OSR implementation when we do not switch. We then evaluate the performance of annotation-guided and automatic GC switching.

VARMAP-Based OSR Performance

We first present results that compare our VARMAP-based OSR implementation to a variation of a commonly used, extant approach to OSR. To implement the latter, we employed the original OSR implementation in JikesRVM. This implementation uses special, unconditional, OSR point instructions to allow OSR at a particular point in the execution. This implementation is used for deferred compilation and method promotion in the original system [38]. We insert OSR points at each gc-safe point (all points at which a GC switch can occur) in each optimized method. An OSR point is a special thread yield point that will trigger on-stack replacement, unconditionally, for the current method. We remove these instructions immediately prior to code generation (after all optimizations) to avoid their execution, since doing so will trigger OSR . By doing so, we are able to measure the impact of OSR on code quality alone.

Figure 3.12 shows the results from this comparison. The y-axis is the percent reduction in execution time enabled by OSR over OSR points (when OSR points are inserted at every gc-safe point during compilation as described above). The Average

bar shows the average across all benchmarks, and Average Spec98 shows the average for only the SpecJVM benchmarks. We gathered results for 25 different heap sizes from the minimum in which the application would run to 8x the minimum at periodic intervals. We report the average over these heap sizes for each benchmark.

Our VARMAP implementation improves overall application execution time by 9% on average across all benchmarks, and by over 10% on average across the SpecJVM benchmarks. jess and mtrt show the most benefit, with improvements of 31% and 20% respectively. For these benchmarks, the original system increases register pressure by extending live ranges of variables. This results in a large number of variable spills to memory. Since we maintain the VARMAP separately from the compiled code, we ensure that live ranges are dictated by the code itself.

Figure 3.13 details the space and compilation overhead of our OSR implementation. Columns 2 and 3 show the compilation time for the clean (reference) JikesRVM system without OSR points and the VARMAP implementation, respectively. Column 4 shows the percentage degradation in compilation time imposed by our VARMAP implementation. Columns 5 and 6 show the space overhead introduced by the VARMAP implementation during compile time (collectable) and runtime (persistent), respectively. On average, our system increases compile time by approximately 26% and adds 132KB of collectable overhead and and 30KB of constant space overhead.

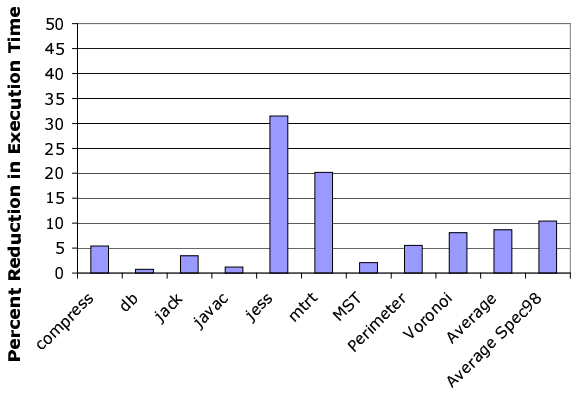


Figure 3.12: Performance of our OSR-VARMAP Implementation in JikesRVM Reference System. Figure shows the average execution time (excluding compilation) performance improvement enabled across heap sizes by our VARMAP implementation over using an extant implementation of OSR – a variation on the OSR points in JikesRVM.

	Compilation Time (msecs)			Space Added (KB)	
Benchmark	Clean	VARMAP	Pct. Degrad.	Compile Time	Runtime
compress	68	79	16.18	14.52	3.16
db	91	117	28.57	24.57	5.26
jack	445	543	22.02	139.67	30.00
javac	1962	2540	29.46	629.94	136.98
jess	504	656	30.16	136.80	29.20
mtrt	595	746	25.38	154.38	33.50
SPECjbb	3515	4431	26.06	42.73	34.14
JavaGrande	2400	2800	17.00	104.86	21.12
MST	50	66	32.00	17.03	3.73
Voronoi	96	129	34.38	62.06	13.49
Avg.	973	1211	26.12	132.66	31.06
Avg. Spec98	611	780	25.29	183.31	39.68

Figure 3.13: Compilation overhead of our VARMAP implementation over the JikesRVM reference system. Columns 2 and 3 are compilation times in milliseconds and column 4 is the percent increase in compilation time. The final two columns show the compilation (collectable) and runtime space overhead, respectively, introduced by our system.

Benchmark	Pct. Degradation No switching vs. Clean
compress	8.98
jess	29.78
db	3.12
javac	12.87
mpegaudio	24.04
mtrt	25.33
jack	6.83
Average	15.85

Figure 3.14: Overhead introduced by the garbage collection switching system when it never switches, over the clean (reference) JikesRVM. The percentage values are averaged over heap sizes. On average, the GC switching system adds a 15% overhead over the clean JikesRVM, *when no switching is triggered*, due to support for on-stack replacement.

We next present results that show the overhead of our VARMAP implementation in our GC switching system when it *never switches* compared to the clean or reference JikesRVM. This is to enable us to evaluate the effectiveness of our VARMAP in reducing the base overhead of the switching system, introduced due to loss of optimization opportunities. The switching system adds an overhead of around 15% on average across applications, when switching is never triggered (see Figure 3.14).

Figure 3.15 shows these results. The numbers show the percentage degradation introduced by the GC switching with the VARMAP implementation (without switching) over the reference JikesRVM image across all measured heap sizes (minimum for each application to large). Average is the average percentage degradation across all benchmarks (5%), and Average Spec98 is the average percent degradation for only the Spec98 benchmarks (<5%). javac has a higher overhead (11%) than other bench-

Benchmark	Pct. Degredation Over Clean
compress	3.71 (285ms)
db	3.09 (662ms)
jack	5.88 (269ms)
javac	11.31 (898ms)
jess	3.06 (104ms)
mtrt	0.62 (81ms)
SPECjbb	3.99 (5908ms)
JavaGrande	3.01 (1944ms)
MST	9.99 (237ms)
Voronoi	5.42 (245ms)
Average	5.01 (1063ms)
Average Spec98	4.62 (383ms)

Figure 3.15: The overhead introduced by the VARMAP version of the GC Switching System over a clean system without GC switching functionality. By reducing the overhead of the Orig-OSR implementation, we are able to cut the base overhead of the GC switching system (the overhead imposed when the system *does not switch*) from 15% to 5%, i.e. the resulting version of the system introduces 5% base overhead over the clean system.

marks due to a larger space (and hence GC) overhead required to store the VARMAP information (see Figure 3.13).

Annotation-Guided GC Selection

To investigate the effectiveness of our GC switching system, we implemented and evaluated annotation-guided and automatic GC selection. In this section, we present results for the former. As we described in Section 3.3, we selected the best performing GC for a range of heap sizes by profiling multiple inputs offline (we list the inputs in

Figure 3.11). The GCs and switch points that we annotate and use are shown in the same table. For brevity, we present results only for the large input.

Our system uses the annotation to switch GCs immediately prior to invocation of the benchmark (at program load time). Our performance numbers *include* the cost of this switch. Moreover, we *specialize* the code for the underlying GC. Our system compiles hot methods with the appropriate allocation routine inlined. In addition, we insert write barriers into all unoptimized (baseline compiled) methods; however, write barriers are inserted into optimized ("hot") methods for generational collection systems. Since our system switches to the annotated GC before the benchmark begins executing, no invalidation or on-stack replacement is required for annotation-guided switching.

As we discussed in Section 3.3, half of the benchmarks that exhibit a switch point. Given such benchmarks and our system's ability to switch between GCs given the maximum available heap size, our system has the potential to enable significant performance improvements since no single collector is the best performing across heap sizes for these programs even for the *same* input.

Figures 3.16, 3.17, 3.18 and 3.19 present performance graphs for representative benchmarks for a range of different heap sizes (x-axis – values are relative to the minimum heap size of the program). The y-axis is program execution time in seconds. For SPECjbb, the y-axis is the inverse of the throughput multiplied by 10^6; we report

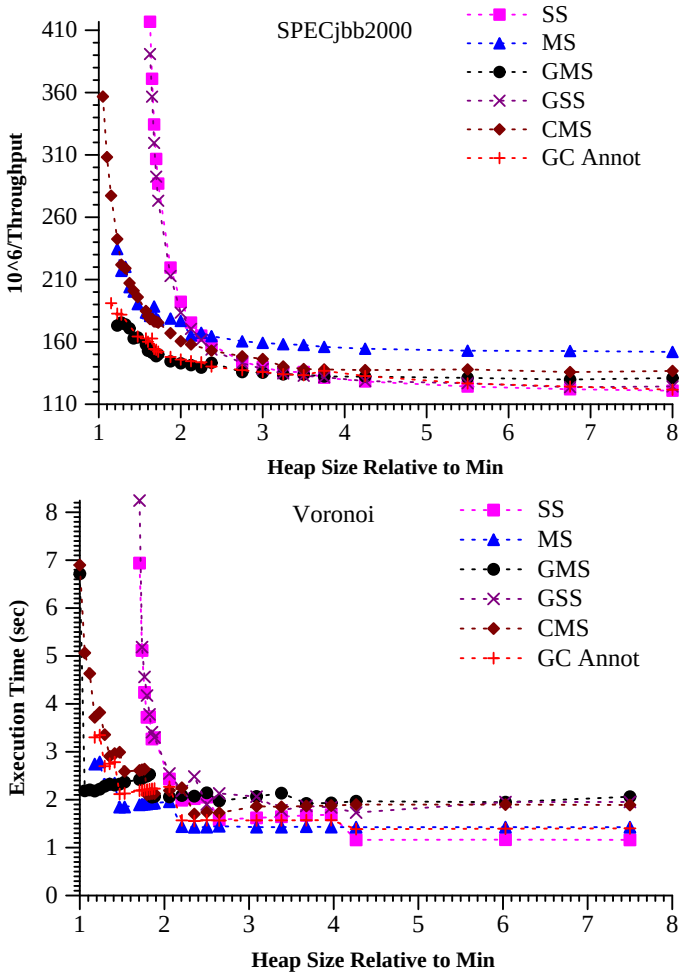


Figure 3.16: Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows two examples with switch points.

this metric to maintain visual consistency with the execution time data, i.e., lower numbers are better. The y-axis value ranges vary across benchmarks.

Each graph contains six curves, one for each of the JikesRVM garbage collectors. These curves represent the performance of the standard JikesRVM garbage

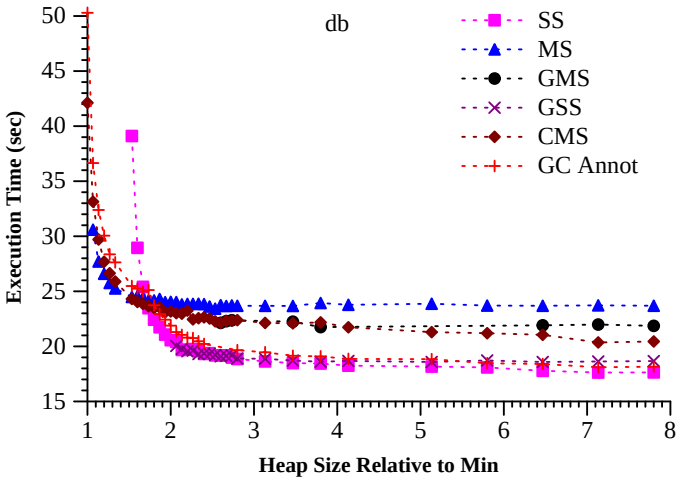


Figure 3.17: Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows an example with switch points.

collectors in the "clean", unmodified, system, in addition to our GC annotation system. The GCs that we evaluate include Semispace (SS), a Generational/Semispace Hybrid (GSS), a Generational/Mark-sweep Hybrid (GMS), a non-generational Semispace/Mark-sweep Hybrid (CMS), and Mark-sweep (MS). The *GC Annot* curve (dashed line with + markers, red if in color) shows the performance of our GC switching system using annotation-guided selection.

The first set of graphs shows three representative benchmarks that have switch points (those that exhibit a change in best performing GC). Our system is able to track the best performing GC for both small and large heap sizes. For example, for db, our system tracks CMS for small heaps and SS for large heaps. As such, for a

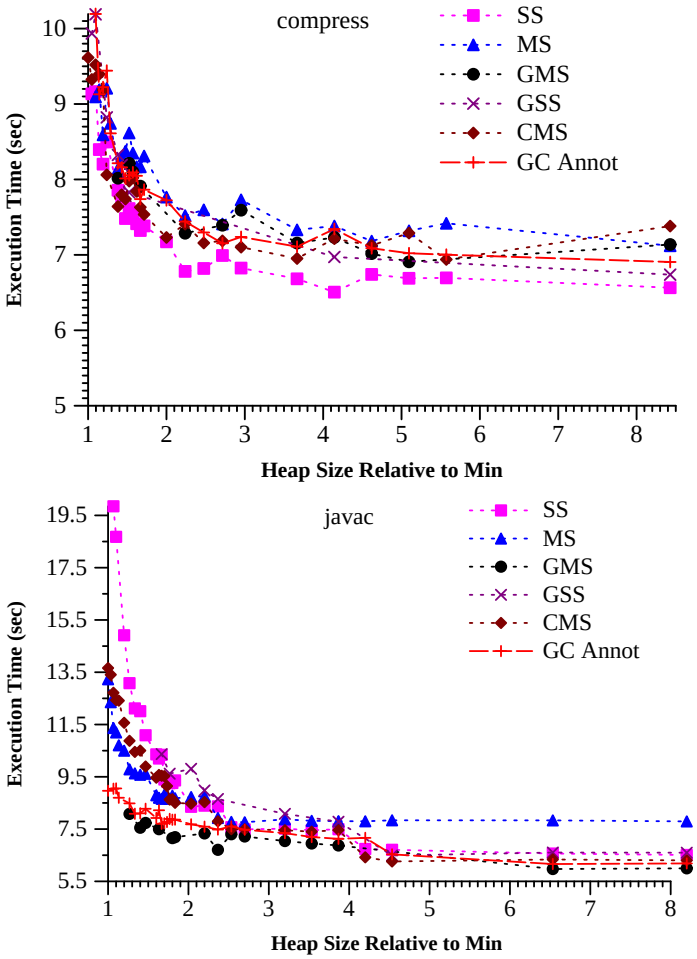


Figure 3.18: Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows two examples without switch points.

single program and input but different resource availability levels, we can improve performance over using *any single collector* for these programs.

The second set shows three representative benchmarks without switch points. For these benchmarks, our system tracks the best performing collector. Notice that the

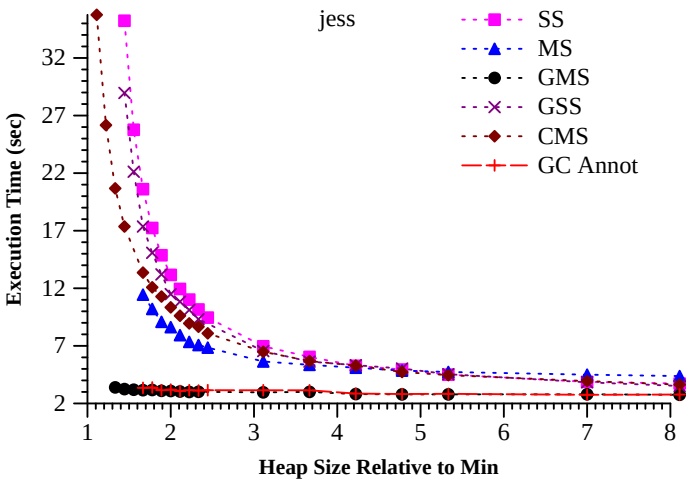


Figure 3.19: Performance comparison between our switching system, GC Annot (dashed line with + marks), and the unmodified reference system built with five different GC systems. The figure shows an example without switch points.

best performing collector differs across programs, e.g., SS performs best for compress and GMS performs best for the others. Since our system uses annotation to guide GC selection and switch dynamically to the best performing GC for each program, it is able to improve performance across benchmarks over any single GC. This becomes more evident when we evaluate this data across benchmarks.

Figure 3.20 and 3.21 summarize our results across benchmarks and heap sizes. Figure 3.20 represents averages for small heaps (minimum for an application to 3x the minimum), and Figure 3.21 represents averages for medium to large heap sizes (from 3x the minimum for an application to 8x the minimum heap size). We present the average difference between our GC switching system and the best performing GC

Average Difference Between Best & Worst GC Systems		
	GCAnnot	
	Small Heaps (upto 3x)	
	Degradation	Improvement
Benchmark	over Best	over Worst
compress	6.65% (484ms)	2.85% (236ms)
jess	4.29% (132ms)	75.01% (10357ms)
db	3.54% (674ms)	8.48% (2108ms)
javac	6.55% (469ms)	27.55% (3626ms)
mtrt	1.31% (81ms)	47.02% (6024ms)
jack	3.34% (156ms)	40.92% (3722ms)
JavaGrande	4.77% (3088ms)	19.11% (17807ms)
SPECjbb	2.59% (3864*10^6/tput)	32.42% (106493*10^6/tput)
MST	3.83% (28ms)	56.80% (1244ms)
Voronoi	8.83% (164ms)	32.13% (1264ms)
Average	4.57%	34.23%

Figure 3.20: Summarized performance differences between our annotation-guided switching system and the reference system for small heap sizes (minimum for an application to 3x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst performing GCs across small heap sizes (the time in milliseconds that this equates to is shown in parenthesis).

at each heap size (column 2) and between our system and the worst performing GC at each heap size (column 3). In parentheses, we show the average absolute difference in milliseconds; for SPECjbb the value in parenthesis is the difference in inverse of the throughput. The table shows that our system improves performance by 34% over selection of the "wrong", i.e., worst performing collector, for small heaps, and by

Average Difference Between Best & Worst GC Systems		
	GCAnnot	
	Large Heaps (3x – 8x)	
	Degradation	Improvement
Benchmark	over Best	over Worst
compress	6.52% (432ms)	3.50% (258ms)
jess	2.04% (60ms)	44.11% (2378ms)
db	2.58% (469ms)	22.83% (5420ms)
javac	4.83% (314ms)	13.40% (1052ms)
mtrt	5.37% (320ms)	27.07% (2364ms)
jack	3.48% (152ms)	14.26% (756ms)
JavaGrande	3.68% (2275ms)	14.93% (11204ms)
SPECjbb	1.77% (2258*10^6/tput)	16.13% (24936*10^6/tput)
MST	4.38% (32ms)	27.38% (318ms)
Voronoi	7.87% (96ms)	30.09% (602ms)
Average	4.25%	21.37%

Figure 3.21: Summarized performance differences between our annotation-guided switching system and the reference system for medium to large heap sizes (from 3x the minimum for an application to 8x the minimum). The table shows the percent degradation over the best- and percent improvement over the worst performing GCs across medium to large heap sizes (the time in milliseconds that this equates to is shown in parenthesis).

21% for medium to large heaps. In addition, the data shows the average performance degradation over optimal selection. This degradation is due to the implementation differences in our system that make it flexible, e.g., write barrier execution in unoptimized code, boot image optimization, switch time (from MS, the default system,

to the annotated system), etc. On average, our system imposes a 4% overhead over optimal GC selection.

Note that the data in these tables does not compare our system against a single JikesRVM GC; instead, we are comparing our system against the *best- and worst performing GC at every heap size*. For example, for large heap sizes for the SPECjbb benchmark, the SS system performs best. For small heap sizes, GMS performs best. In this case, to compute percent degradation, we take the difference between execution times enabled by our system and the SS system for large heap sizes, and our system and the GMS system for small heap sizes.

We also collected the same results for when we omit Mark-Sweep (MS) collection. MS works well for small heaps but is thought to implement obsolete technology. On average across benchmarks and heap sizes, our system imposes 3% overhead over the best performing GC at each point. In addition, our system reduces the overhead of selecting the worst performing collector by 21 – 34% (depending on the heap size). Interestingly, when MS is not available in the system, the average degradation *decreases*. This is due to the fact that MS is the best performing collector in a number of cases in which small and medium sized heaps are used.

Figure 3.22 presents the percent degradation over *always using the Generational/Mark-Sweep Hybrid (GMS)*. GMS is thought to be the best performing, JikesRVM GC – it is the default collector in JikesRVM version that we extended. However, our data

Benchmark	GC Annot: Average Degradation Over Generational Mark-Sweep
compress	-0.37% (-28ms)
jess	2.82% (85ms)
db	-14.17% (-3122ms)
javac	5.19% (373ms)
mtrt	2.32% (78ms)
jack	3.22% (147ms)
JavaGrande	-0.19% (-87ms)
SPECjbb	0.95% ($1.72*10^6$/tput)
MST	-44.66% (-827ms)
Voronoi	-11.88% (-241ms)
Average	-5.68%

Figure 3.22: Percent degradation of our system over the widely used GMS collection. The negative values indicate that on average across heap sizes, our system improves performance over GMS.

shows that it does not work well for all programs for all heap sizes. Our system enables a 6% improvement (a negative degradation) over always using GMS across benchmarks and heap sizes. This improvement varies across inputs: 14% and 12% for db and Voronoi, to almost 45% for MST. Note, however, that MST is a very short running program – small differences in execution time (800ms) translate into very large percent differences. The improvement in db translates to a benefit of over 3 seconds.

Overall, these results indicate that our framework is able to achieve performance that is similar to the best performing collector (in terms of both execution performance and compilation overhead) by making use of the annotations to guide dynamic switching between GCs. Moreover, when there is a switch point for programs, our system can enable the best performance on average over any single GC for that program. For cases in which there is no crossover between optimal collectors, our system maintains performance similar to that of the reference system. However, since the optimal GC varies across benchmarks, our system is able to perform better than any single GC across benchmarks.

Automatic Switching

We next evaluate the effectiveness of automatically switching between GCs using online program behavior and simple heuristics. Automatic switching requires the use of method invalidation and OSR to maintain correctness given the use of aggressive specializations: including/avoiding write barriers and inlining allocation routines – for the currently available, underlying GC. Our system employs our new version of OSR to enable both high performance and correctness.

The automatic switching scenario that we investigate addresses what happens when there is suddenly a loss of memory availability, i.e., the OS reclaims memory from the JVM for use by another, high-priority, application. In such a case, automatic

switching can avoid OutOfMemory errors (or prevent excessive paging) by switching to a GC that works well when resources are constrained. We investigated the case in which memory was reduced to a point that the program can still make progress. For such cases, by switching to a more appropriate GC, we can reduce the overhead of garbage collection and improve performance.

We consider the situation in which after program startup, the OS reclaims memory such that the resulting heap size is twice the size of the reserved space (live data) following a garbage collection. We start with a maximum heap size of 200MB. We trigger heap resizing when the program steady state begins – which we approximate by 100 thread switches (we use 500 for SPECjbb since it is a longer running program). The switching system decides to switch when the GC load (defined in section 3.4) remains high for multiple GC cycles (we use three in the results). In addition, the system also switches when it observes that GCs are being triggered too frequently, measured as the duration for which application threads execute between successive garbage collections (we choose 300ms as the minimum application duration observed over 3 GC cycles).

We present the performance of this scenario in Figure 3.23. Columns 2 and 3 show the time in seconds for execution for the clean (Base) system and our automatic switching system (including all overheads). Column 4 shows the percent improvement enabled by our system. On average, our GC switching system can improve the

performance of the program given dynamically changing resource conditions by over 21%. For the SpecJVM98 benchmarks, we improve performance by 29% on average. Interestingly, for some benchmarks, we found that Generational Mark-Sweep (GMS) incurs more garbage collections compared to always executing the application with Semispace. Yet, switching to GMS benefits the application since the total GC time is less compared to Semispace, since on average, a single GMS collection runs for a very short duration (as low as 9 milliseconds) compared to a typical Semispace collection (150 to 200 milliseconds). compress and MST do not allocate enough for a switch to be triggered. The right half of the table shows the OSR statistics. Column 5 is the number of OSRs, column 6 is the total OSR time in milliseconds, and column 7 is the heap size following the system memory reclamation.

In summary, automatic GC switching has the potential for enabling the application to make progress and avoid OutOfMemory errors if resources become constrained during program execution. In addition, it improves performance under such conditions by switching to a GC that imposes less GC overhead. Should memory availability be restored, our system can switch to a collector that performs well for large heap sizes, e.g., SS. Given the ability to dynamically and efficiently switch between competing collection systems, we now have the ability to consider other mechanisms (e.g., program phase and data locality behavior) for deciding when to

Benchmark	Base	Autoswitch	Pct. Impr.	# OSRs	OSR Time (ms)	Heapsize (MB)
compress	7.65	7.65	0.00	--	--	60
jess	7.23	3.89	46.20	10	28.46	28
db	31.29	23.16	25.98	1	1.39	24
javac	11.73	10.72	8.61	10	22.45	47
mtrt	24.77	9.11	63.22	2	58.35	24
jack	7.53	5.36	28.82	4	6.50	32
SPECjbb	175.10	158.70	9.71	0	0.00	100
JavaGrande	102.09	76.80	24.76	1	1.58	24
MST	0.94	0.94	0.00	--	--	100
Voronoi	4.37	3.94	9.15	2	3.00	60
Average	37.27	30.03	21.65	4	15.22	50
Average Spec98	15.03	9.98	28.81	5	23.43	36

Figure 3.23: Performance of automatic switching when memory resources are suddenly constrained. Columns 2 and 3 show the time in seconds for execution for the clean (Base) system and our automatic switching system (including all overheads). Column 4 shows the percent improvement enabled by our system. The right half of the table shows the OSR statistics: number of OSRs, total OSR time in milliseconds, and the heap size following the memory reclamation by the system.

switch and to which GC we should switch to. We plan to investigate such techniques in future work.

3.6 Related Work

Two areas of related work show that performance due to the GC employed varies across applications and that switching collectors dynamically can be effective. In [67, 78], the authors show that performance can be improved by combining variants of the same collector in a single system, e.g., mark-and-sweep and mark-and-compact. and semispace and slide-compact In [81], the authors show that coupling compaction with a semispace collector can be effective. No extant system, to our knowledge, provides

a general, easily extensible framework that enables dynamic switching between a number of completely unrelated collectors.

Other related work shows empirically that performance enabled by garbage collection is application-dependent. For example, Fitzgerald and Tarditi [40] performed a detailed study comparing the relative performance of applications using several variants of generational and non-generational semispace copying collectors (the variations had to do with the write barrier implementations). They showed that over a collection of 20 benchmarks, each collector variant sometimes provided the best performance. On the basis of these measurements they argued for profile-directed selection of GCs. However, they did not consider variations in input, required different prebuilt binaries for each collector, and only examined semispace copying collectors.

Other studies have identified similar opportunities [5, 101, 82]. IBM's Persistent Reusable JVM [55] attempts to split the heap into multiple parts grouped by their expected lifetimes, employs heap-specific GC models and heap-expansion to avoid GCs. It supports command-line GC policies to allow the user to choose between optimizing throughput or average pause time. BEA's Weblogic JRockit VM [11] employs an adaptive GC system that performs dynamic heap resizing. It also automatically chooses the collection policy to optimize for either minimum pause time or maximum throughput, choosing between concurrent and parallel GC, or generational and single-spaced GC, based on the application developer's choice. BEA's white-

paper [11], however, describes the system at a very high level and provides few details or performance data. We were unable to compare our system against the JRockit, due to its proprietary nature. To our knowledge, no extant research has defined and evaluated a general framework for switching between very diverse GC systems, such as the one that we describe. In addition, our automatic switching heuristic, albeit simple, requires no user intervention and achieves considerable performance improvement.

On-stack replacement (OSR) was initially conceived of by the researchers and engineers of the Self-91 system [22]. The system employed OSR to defer compilation of uncommon code until its initial execution, to increase optimization opportunities, and to reduce compiled code space and compilation overhead. The authors in [47] extended OSR to enable dynamic de-optimization of optimized code to facilitate debugging; [47] describes the complete OSR implementation in Self. Our OSR technique is similar to this one since optimized code is replaced and OSR occurs *lazily* as control is transferred back to executing methods (via return instructions). However, in this prior work, de-optimization can occur (and hence debugging can commence) only at two points in a method: method prologue and loop back-edges. As such, state extraction is needed only at these points. Our system must extract state at these points as well as all call sites (which include allocation sites) to enable OSR to occur at any point in a method at which control is transferred to another executing thread.

In Self-93 [48], the Self group used OSR to improve execution performance within an adaptively optimizing runtime system. The system recompiled hotspots and used OSR to enable optimized execution of currently executing unoptimized methods; this process is also called *method promotion*. State extraction for method promotion is somewhat trivial since the method being replaced is unoptimized and all variable values can be easily identified.

To enable deferred compilation, the Self system inserts unconditional calls that invoke the OSR process at points that guard paths to uncompiled code. We refer to these calls as OsrPoints. Recently, Fink et.al. [39] presented the implementation and empirical evaluation of unconditional OSR in the Jikes Research Virtual Machine from IBM T.J. Watson Research Center. The unconditional OSR instruction, i.e., OSRPoint, implementation is based on the Self implementation and is similar to other deferred compilation systems [88]. Fink et al use the system to implement profile-guided deferred compilation and method promotion.

We extend this prior JikesRVM OSR implementation in this work. OsrPoints are a restricted and simpler (in terms of their implementation) version of the general-purpose OSR that we describe herein. These prior approaches require that the compiler insert explicit, “pinned”, instructions at the point at which invalidation and OSR *must* occur. Since our goal is to use OSR to correct for specialization assumptions invalidated by *external events* such as class loading, a change in the implementation of

MRE services, or a user event, we cannot use unconditional OsrPoints – since we do not know when such events will occur. Instead, we extend the JikesRVM OSR system to enable state collection at *any* point at which assumptions *might* be invalidated.

3.7 Summary

Managed runtime environments (MREs) are ubiquitous and provide safe and portable mechanisms for the execution of type safe code. MREs typically run diverse type of applications ranging from scripting engines to databases and application servers. Dynamic memory management, i.e. garbage collection (GC) is a key component of MREs. Garbage collection plays an increasingly important role in next generation Internet computing and server software technologies.

The performance of collection systems is largely dependent upon application execution behavior and resource availability. In addition, the overhead introduced by selection of the "wrong" GC can be significant. To overcome these limitations, we have developed a framework that can automatically switch between GCs without having to restart and possibly rebuild the execution environment, as is required by extant systems. Our system can switch between collection strategies *while* the program is executing.

We present specialization techniques that enable the system to be very low overhead and to achieve significant performance improvements over traditional, non-switching, virtual execution environments. We describe a novel implementation of on-stack replacement (OSR) that can enable efficient replacement of executing code at any point in the program at which a GC (and thus a GC switch) can occur.

We also present two techniques that exploit the efficient GC switching functionality. In particular, we describe and present the effectiveness of annotation-guided (based on offline profiling) and automatic (based on online profiling) switching. We empirically evaluate our system using a wide range of heap sizes, benchmarks, and scenarios.

The text of chapter 3 is in part a reprint of the material as it appears in The Elsevier Journal of Systems and Software (JSS), Volume 80, Issue 7 (2007). The dissertation author was the primary researcher and author and the co-authors listed directed and supervised the research that forms the basis for these this chapter.

Chapter 4

Scalable Memory Management for Multi-Tasking Managed Runtime Environments

The second part of this dissertation focuses on memory management for multi-tasking persistent MREs. MREs commonly execute a single program with a single MRE instance, and rely on the underlying operating system to isolate programs from each other for security, as well as for resource management and accounting.

Unfortunately program isolation at the granularity of the virtual machine can significantly restrict the performance of MREs that execute multiple, independent, programs concurrently. This execution model duplicates effort across MRE instances, since it prohibits sharing of MRE services and internal representations, memory, code, etc., across programs. Such redundancy increases startup time and memory consumption and degrades overall system performance and scalability.

A multitasking implementation of an MRE can address these problems while maintaining portability, mobility, and type-safety. We focus a state-of-the-art implementation of a multi-tasking MRE, Sun Microsystem Labs' Multi-tasking Virtual Machine (MVM) [29] that executes multiple programs within a single operating system process. Co-locating programs in the same address space simplifies the virtual machine implementation through sharing of the runtime representation of programs and dynamically compiled code. Such sharing also avoids duplicated effort across programs (e.g. loading, verification) and amortizes runtime costs, such as dynamic compilation, over multiple program instances. Prior work on the MVM [27], shows how a multitasking design reduces startup time and memory footprint, and improves performance over a single-program MRE approach.

However, prior work on multi-tasking MREs does not address the performance of concurrent workloads, i.e., multiple applications executing *simultaneously* . Multi-tasking MREs are designed to run multiple applications simultaneously, and no prior work has shown conclusively that multi-tasking can outperform single-tasking, while maintaining similar memory footprint for concurrent application execution. In addition, the design of the state-of-the-art MVM MRE suffers from several drawbacks, such as lack of performance isolation in the memory management subsystem and imprecise tracking of heap resources. Garbage collection triggered by any application pauses all applications. Further, a multi-tasking MRE is designed to have a lifetime

that is longer than any single application. The MVM suffers from the inability to free a terminated application's resources without having to pause and garbage collect all applications executing in the MRE.

We present the *Multi-Tasking Memory Manager* (MTM) which combines multi-tasking memory management techniques to provide performance isolation, per-application, scalable generational garbage collection, GC-free reclamation of terminated applications' resources, per-application control of memory subsystem parameters, while constraining memory footprint. We have prototyped MTM in Sun Labs' state-of-the-art Multi-tasking Virtual Machine (MVM). We show that MTM enables multi-tasking to outperform single-tasking MREs and that multi-tasking is a viable approach for executing concurrent applications. We describe our approach in detail below.

4.1 Application-Aware Memory Management for Multi-Tasking Managed Runtime Environments

The current MVM system [92] implements a simple memory management system in which a single heap and management policy is shared across all applications. Such sharing does not isolate applications from interfering with one another (in terms of performance), and restricts the scalability of the system. Moreover, there is no per-application control over GC parameters or reclamation of heap resources upon appli-

cation termination without requiring an expensive, full heap GC. Extant multi-tasking approaches (e.g. [26]), that do not employ MRE support, impose similar restrictions. An alternative approach is to assign a separate heap space (and possibly different GC policies) to each application. Using such an approach complicates the memory management system, restricts the opportunistic use of reserved idle memory by other applications, and can limit the number of concurrent applications that the system can support.

We present a design that addresses these challenges for Sun Microsystems' MVM [92]. Key to our design is an organization of the heap that enables (i) per-application performance isolation for the memory management system, (ii) independent allocation and collection of young objects, and (iii) GC-free memory reclamation upon application termination.

The design follows a hybrid approach that divides the heap into application-private and shared sections, so that we confine a majority of GC activity to application-private sections. This hybrid organization of the heap works particularly well with generational GC algorithms that divide the heap into multiple generations. Generational GCs segregate objects by age and concentrate their GC efforts on the youngest generations (i.e., the generations holding the youngest objects) by exploiting the weak generational hypothesis [94], which states that most objects die young.

In our implementation, the heap consists of multiple *independent young generations* (one per running application), and a single old generation that all applications share. When an application enters the system, it is given a private young generation that the system sizes according to parameters specified by the application. An application allocates primarily from its young generation. When this area is full, the system performs a *minor collection for that application*. During a minor collection, the GC system moves (promotes) mature live objects to the shared old generation.

The shared old generation efficiently tracks the regions that each application consumes using *old generation regions*. An application uses its old generation regions both for object promotion during minor collections and for direct (pre-tenuring) allocation of objects. Per-application old generation regions provide numerous advantages – they cluster objects of the same application together in the shared old generation, they ease accounting of space consumed by applications in the shared generation, they enable immediate reclamation of old generation space without garbage collection upon application termination, and they help limit the amount of old generation space that the system must scan to identify roots during minor collection. In addition, by combining per-application young generations with old generation regions, we eliminate interference between mutators and collectors of different applications. As a result, our system is able to perform minor collection for an application, concurrently with the execution of mutators of other applications.

Although simple, the approach of sharing a generational heap between dynamically varying numbers of independent applications presents several problems. First is the absence of *performance isolation* with respect to garbage collection. That is, garbage collection affects all applications at once, and has a cost proportional to the live objects of all applications. A second problem is the inability to immediately reclaim the heap space consumed by an application, upon its termination. Resource reclamation requires a full garbage collection, which affects all applications. Both problems adversely impact scalability and response time.

The following section presents a generational garbage collection system that attempts to better address the requirements of MVM with a combination of three features – per-application independent young generations, per-application old generation regions, and application-concurrent scavenging.

4.1.1 Hybrid generational heap

The first element of the design builds on [27], by providing each application with a private young generation, while sharing a single old generation between all applications. This hybrid approach attempts a compromise between sharing the heap between all applications and giving each application an independent heap. There are several reasons for this choice.

First, the young generation is typically much smaller than the old generation. Thus, having one per application young generation and sharing the tenured space makes better use of heap resources, by avoiding committing too much memory per application, and unnecessarily limiting the degree of multi-tasking. Old generation space is allocated to an application on demand, either during minor collection, or when pre-tenuring objects.

Second, the vast majority of allocations and most garbage collections occur over the young generation. Thus, an independent young generation shields an application from most heap-related interference, especially varying allocation rate, tenuring decisions, and interleaving of objects from different applications. Also, minor collection pauses are proportional to the live set of objects of a given application, as opposed to all applications.

Third, key parameters for generational garbage collection, such as young generation size, age-based tenuring policy, etc., can be controlled on a per-application basis. This enables users to specify an appropriate application-specific set of tuning parameters for each application.

Figure 4.1 depicts our layout for per-application young generations. A table, called the young generation virtualizer, maps the application identifier to the corresponding young generation. Each young generation has the same layout as in the original HotSpot JVM, with the notable exception that a young generation may con-

sist of several discontinuous regions of memory. Specifically, space for young generations is allocated from a pool of fixed-sized chunks, the size of which is parameterizable and set at 2MB by default. On startup, an application is allocated an integral number of chunks corresponding to the size of the young generation requested (or the default if none is specified). The chunk manager attempts to allocate contiguous chunks when possible, otherwise, it assigns additional edens to the young generation, one per region of contiguous chunks allocated from the pool (similarly to the surplus memory in [27]). The pool manager may re-arrange the chunks allocated to a young generation to reduce the fragmentation of its eden. Such re-arrangement takes place as necessary following minor collection, when all the live objects of the eden space have been evacuated. This organization also allows us to dynamically change the size of a young generation at runtime.

The to and from spaces are typically much smaller than the eden space. For simplicity, the current prototype limits their size to that of a single chunk.

As in the original HotSpot JVM, threads are assigned one or more thread local allocation buffers (TLABs) so that they can allocate objects without synchronization with other threads. The TLAB of a thread is allocated from the eden of the young generation of the thread's application.

Per application young generations provide some degree of performance isolation – the copying cost of scavenging is proportional to the number of live objects of the

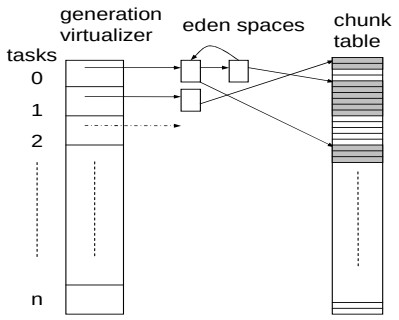


Figure 4.1: Application independent flexible young generations. A generation virtualizer maps applications to young generations. Each generation comprises one or more eden spaces, each of which consists of an integral number of contiguous chunks allocated from a pool. Eden spaces of an application are linked together. Chunks can be added or removed dynamically.

application that triggered the scavenge; further, only mature objects of that application are promoted to the shared old generation.

However, per application young generations alone are insufficient for complete performance isolation. All applications must still be stopped at a safepoint in order for the scavenger to have a consistent view of the old generation. In particular, consistency of the remembered set of references to young generations must be guaranteed, in order to precisely locate references from the old generation, to the young generation being scavenged. Note, however, that applications are stopped at the safepoint only for the duration of the scavenge of the live objects of a single application, which improves over a design that shares a single young generation between applications.

Another concern is that per application young generations do not enable immediate reclamation of all heap space consumed by a terminated application. The young

generation can only be reclaimed when there are no longer any references to it from the old generation. Otherwise, it may lead to situations where an obsolete pointer from the old generation may be mistaken for a valid pointer if the reclaimed space has been re-allocated for the young generation of another application. For this reason, young generation space can only be freed once all such references have been cleared. This can be done opportunistically at any scavenge, while scanning the remembered set. In addition, space consumed by a terminated application in the old generation can only be reclaimed upon a full collection of the old generation.

To address the problems listed above, we complement per application young generations with old generation regions. Regions allow instantaneous, collection-less, reclamation of all heap space (i.e., both young and old) consumed by a terminated application. Regions also help to simplify synchronization issues towards efficient support for application-concurrent scavenging.

4.1.2 Per-application Old Generation Regions

Immediate, collection-less reclaiming of the heap space used by a terminated application can be obtained by precisely tracking old generation regions in which objects allocated by each application reside. With this knowledge, young generation collection can ignore all regions of the old generation that do not contain objects of the application being scavenged, since these are not required to determine roots for

collection. Since no regions of old generation that may contain obsolete references to young generations of terminated applications will be scanned, young generations of terminated applications can be re-used immediately, without GC.

The old generation space used by a terminated application can be re-used immediately without any collection as well. The only references to regions used by a terminated application originate from the tables used to mediate access from the shared part of the runtime representation of classes stored in the permanent generation, to their application-private parts located in the old generation. Thus, the regions corresponding to a terminated application can be immediately re-used, if the GC ignores entries of the tables corresponding to terminated applications. This, however, prevents re-use of the identifiers of terminated applications. These identifiers will eventually must be reclaimed by cleaning corresponding entries in the global application table. The cleaning of these entries can be done opportunistically on the next GC that requires scanning the application table, or by a separate background thread. Note that cleaning itself does not require any synchronization with applications.

Precisely identifying which regions of the shared old generation hold objects of a terminated application is key to the collection-less reclamation of the heap space used by the terminated application as described above. Tracking individual objects would likely be prohibitively expensive. Instead, we propose per-application old generation regions ,i.e., a contiguous region of the old generation assigned to a application. Old

generation regions are primarily used during scavenging of the young generation of an application when promoting young objects to the old generation. They are also used for the occasional direct allocation of objects in the old generation, either because the object does not fit in the young generation, or as a result of a pre-tenuring decision. For example, as described previously, the application-private representation of a class is always pre-tenured. The size of a region can be application-specific and adjusted dynamically. It is generally chosen to satisfy several scavenges (promotions). Allocation in a region involves increasing a cursor to the first free byte in the region (bump-pointer). When mutator threads allocate in a region, synchronization between threads is required, since the region of an application is shared between all threads of the application.

Figure 4.2 illustrates old generation region management. Each application is associated with a current region and a list of full regions. An initial region is allocated to an application at startup, prior to the first allocation by the application. When a region is full (typically during a scavenge), its address is recorded in the application's list of full regions, and a new one is provided to the application.

If an object does not fit in an old generation region, space is allocated directly from the old generation, either from a previously freed region, or from the free space at the end of the old generation (beyond the last region). In both cases, the object is recorded as a full region in a list corresponding to the application performing allocation. The

list of full regions, thus, precisely tracks regions of the old generation used by an application.

When an application completes, its application identifier is added to a list of applications whose application table entries can be freed and re-used. The application's current region and full regions are added to a global list of free regions, and become immediately available for re-use by other applications. Young generation chunks of the application are returned to the global pool, and are immediately available for re-use by the young generations of other applications (see Section 4.1.1).

Adjacent free regionsPare coalesced in a single region. Free regions at the end of the old generation are removed from the list and the pointer to top of the old generation is updated accordingly, as illustrated in Figure 4.3. Apart from limiting the space overhead of tracking regions, coalescing can increase the size of contiguous free region areas, consequently limiting fragmentation and further reducing the frequency of full GC.

regions have several interesting properties. First, they improve isolation between applications, since most allocation in the shared old generation is performed from a region that is private to one application, eliminating a point of interference between applications. Second, they efficiently keep track of the old generation space used by applications. Tracking is relatively inexpensive and only involves adding a region to a list of full regions when a region is full, or when an object larger than the current

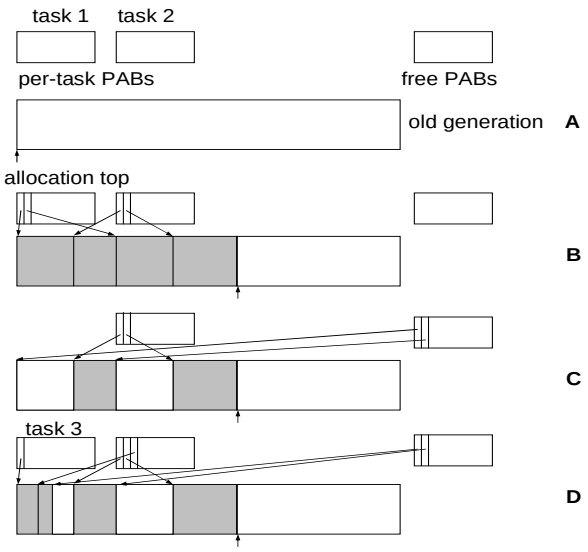


Figure 4.2: Example of region management & tenured space reclamation at application termination without a full GC. (A) Initial configuration. (B) Both applications 1 & 2 have performed promotions and their respective full region list are now non-empty. (C) Application 1 terminates and its set of full regions is added to the global free list. (D) Application 3 enters the system and application 2 & 3 start using space allocated from the region free list.

region capacity is allocated. This precise tracking enables collection-less reclamation of both young and old generations space used by terminated applications. Further, it optimizes the identification of references from the old generation to a particular young generation (as will be described later). Last, it enables precise accounting of space consumed by applications.

Maintaining regions Across Major GCs

Reclamation and reuse of regions mitigates full heap GC, but is not a replacement for it. The old generation may fill up eventually, requiring collection. A sliding

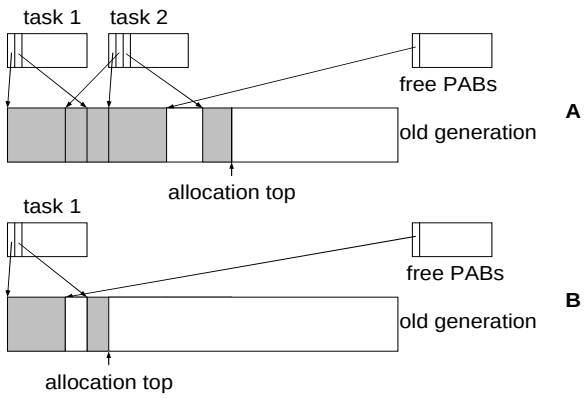


Figure 4.3: Example illustrating shrinking of old generation footprint upon application termination.

mark-compact collector is used for the old generation. The collector may reclaim garbage in regions and compact live objects inside regions, thus invalidating their original boundaries. Consequently, old generation collections may require adjustment to regions boundaries. The following describes how this adjustment is performed (Figure 4.4).

The old generation mark-compact GC is a standard 4 phase compacting collector [74] involving the following phases.

- Mark live objects.
- Compute new addresses for live objects.
- Scan objects and adjust references to point to the new locations.
- Relocate (copy) objects to their new locations.

Adjustment of region boundaries can be performed between the second and third phases. During the second phase (computing new addresses), the GC stores the new address for a live object that will be relocated, in the object header. To compute new boundaries for a particular region, we locate the address of the first live object in the region. If no live object is found, this region can be dropped from the corresponding application's list. If a live object is found, we read its new address from the header, which now becomes the new start of the region. To adjust the end of a region, we note that the first live object beyond the end of the region would be moved to a location right after the new end of the region. The new end of the region is therefore the new location of the first live object past the current end.

Note that locating the first live object from either the start or end of a region can be expensive. However, we make use of an optimization that the existing garbage collector itself uses to quickly skip over dead objects. During the second phase of mark-compaction, the GC records the address of the next live object in the header of the first dead object in a group of contiguous dead objects. In the best case, the current boundary of an old generation region is the first dead word in a group of dead objects. However, this may not always be the case, hence, we may need to iterate over successive dead objects until we find the next live (GC marked) object. To avoid excessive scanning, it may be necessary to limit the number of dead objects scanned, and discard the region entirely if this number is over a threshold. In practice, we

```
adjust_promotion_area(PromotionArea pa) {
 pa.start = adjust(pa.start);
 pa.end = adjust(pa.end);
 if(pa.start == pa.end) pa = NULL;
}
Word* adjust(Word* q) {
 if(q < first_dead) //GC maintains address of
        return;     //first dead object found in
                      //phase 2 of mark-compact
 new_q = NULL;
 while(q < end) { //end here is the end of old gen before GC
        new_q = forwaring_word(q);
        if(is_gc_marked(q)) {
                return new_q; //forwarding word is new location
        } else {
                if(new_q != NULL) {
                //fast case in determining next live object
                //q happens to be the first dead object of a
                //clump of dead objects: next live object is new_q
                q = new_q;
                } else {
                //q happens to be in the middle of a clump of dead
                //objects. Iterate till we find the next live object.
                q = q + size(q);
                }
        }
 }
 //we reached the end without finding the new location for q
 if(q > new_top)      //new_top is the end of the last live
         return new_top; //object after GC
 return NULL;
}
```

Figure 4.4: Region adjustment at full GC. *pa* is the region to be adjusted.

find that this overhead is not excessive. Note that discarding regions does not affect correctness.

Optimizing Scavenging

Scavenging uses a card table [21, 50, 45] to identify references from the shared old generation to per-application young generations, in order to identify reachable young objects. In the presence of a large number of dirty cards belonging to different applications, scanning the entire set of dirty cards at each scavenge might prove expensive. The existing card table implementation does not associate cards with applications and hence, every scavenge requires scanning all dirty cards. Having mutators record ap-

plication information in cards would add an additional cost to the write barrier, thus negating an important advantage of using a card table. In addition, extra space per card would be needed to record a application identifier, or a list of application identifiers.

Our scheme of tracking per-application old generation usage via regions can be readily used to scan dirty cards of only the application initiating young generation collection. This substantially reduces the number of cards being scanned. During card table scanning, we only iterate over the dirty cards that correspond to the list of regions for the application that initiates GC.

4.1.3 Application-Concurrent Scavenging

By combining independent young generations and old generation regions, we implement a mechanism that enables mutator activity and minor collections to be performed concurrently. We refer to this mechanism as *mutator-concurrent scavenging*.

Mutator-concurrent scavenging requires maintaining consistency while scanning of the old generation during promotion. In order to maintain a consistent view of the old generation, changes to the old generation during direct allocation must not affect old generation objects accessed during scavenging. This requires that both object allocation and initialization of the object be done atomically in order for the collector to only trace objects with valid class information. Guaranteeing the atomic behavior

of these two operations cannot be done efficiently with non-blocking synchronization (in contrast to allocation alone which can be implemented with a single compare-and-swap operation, i.e. `cas`). Other synchronization mechanisms would impose a prohibitive overhead on allocation.

regions provide a synchronization-free solution since we need only to scan application-private regions during scavenging. Other applications may directly allocate in their own private regions without affecting minor collection.

Key to mutator-concurrent scavenging is a modified synchronization mechanism that only pauses threads that belong to the application that triggers collection (the *trigger* henceforth), during scavenging. This process first obtains a global *Threads_lock* so that no new threads can be started, or existing threads terminated while the runtime is negotiating a safepoint. We then count the number of threads belonging to the trigger that are running, and iterate until this number reaches zero.

In the MVM, threads periodically poll (access) a constant reserved address that does not belong to the application heap. This address lies on a protected page and accessing this page results in an exception. The exception handler is responsible for blocking threads for a safepoint operation. We make polling application-aware by making threads access an application-private polling page. When a non-global safepoint is initiated, we set only the polling page for threads belonging to the trigger

```
begin_per_task_safepoint {
  Threads_lock->lock(); //no threads should terminate or start
  Safepoint_lock->lock(); //only 1 safepoint at a time
  ∀ t ∈ Threads
        if wants_safepoint(t) { //t belongs to initiator
                ++running;
                protect(t.polling_page);
        }
  while(running > 0) {
         ∀ t ∈ Threads
                if wants_safepoint(t) {
                        //wait until t is waiting on
                        //Scavenge_lock
                        if(!is_running(t))
                                --running;
  }
  //safepoint reached
  Safepoint_lock->unlock();
  Threads_lock->unlock();
}

end_per_task_safepoint {
  Threads_lock->lock(); //no threads should terminate or start
  Safepoint_lock->lock(); //only 1 safepoint at a time
  ∀ t ∈ Threads
        if wants_safepoint(t) { //t belongs to initiator
                unprotect(t.polling_page);
                t->restart();
        }
  Scavenge_lock->notify_all(); //wake up all threads waiting
                               // on the Scavenge_lock
  Safepoint_lock->unlock();
  Threads_lock->unlock();
}
```

Figure 4.5: Per-application safepointing; *begin_per_application_safepoint* initiates a safepoint for a single application and *end_per_application_safepoint* ends it and resumes mutators for that application.

to an address that corresponds to a protected page. An exception will be triggered for these threads when they poll for a safepoint.

The exception handler causes threads to wait on a *Scavenge_lock*, which will only be released when scavenging is complete. Note that only threads belonging to the trigger will wait on the *Scavenge_lock*. When all such threads are paused, the number of threads running drops to zero, and GC commences. Threads belonging to other

applications may continue to allocate, however, they may not perform a GC while the current GC is in progress. Releasing a safepoint is the reverse of this process. The private polling page for blocked threads is set to an address belonging to an unprotected page, and the *Scavenge_lock* is released. This process is illustrated in Figure 4.5.

4.1.4 Evaluation

To evaluate our extensions to MVM memory management, we performed a number of empirical experiments. We gathered our results using a dedicated dual 1.5GHz UltraSPARC system, running Sun Solaris 10. The MVM implementation that we extended in this work is based on Hotspot 1.5. We present results for a number of SpecJVM98 [85] and Dacapo [31] benchmarks.

To evaluate the performance of our system, we first present throughput and response time for short running applications, when executing concurrently with a GC-intensive program. We then consider throughput as well as the overall performance of concurrent, homogeneous applications. Finally, we analyze the impact of our techniques on the number of GCs that the system performs as well as the time spent in GC.

In the first set of results (Figures 4.6 and 4.7), we show the throughput and response time improvement enabled by independent young generations and regions

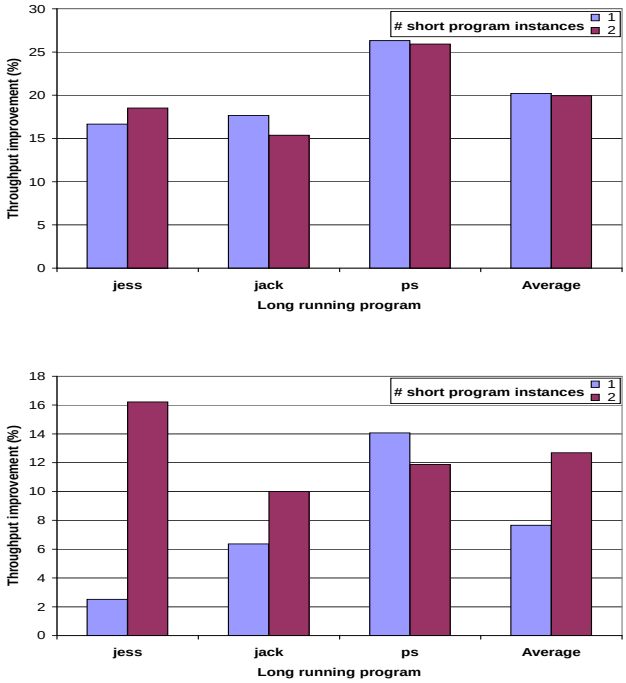


Figure 4.6: Throughput improvement enabled by independent young generations & regions for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications: `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program.

over a system with a shared young generation. In this set of experiments we execute multiple serial instances of a short running application, concurrently with a single instance of a GC-intensive application in a fixed time interval. The goal is to measure the number of serial instances of the short running application that can be executed with the shared young generation system, versus the number of instances of the same application executed with an implementation that includes independent young generations and regions. We also report response time (average application time) for the

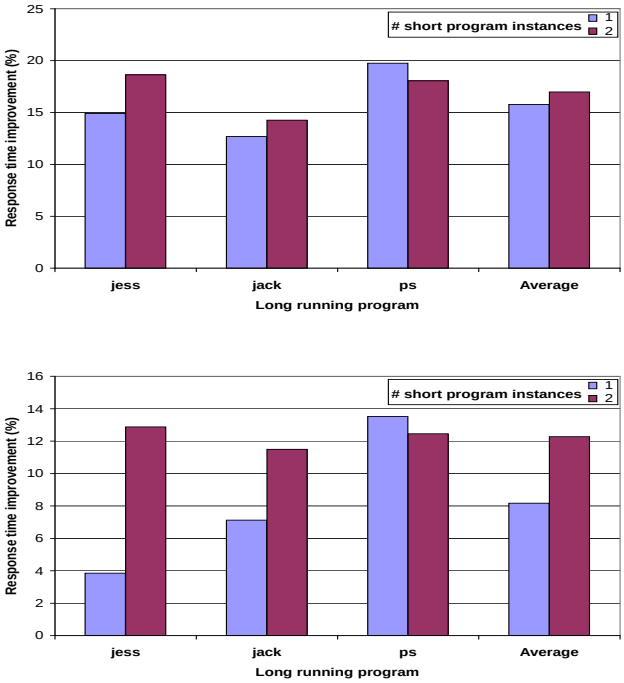


Figure 4.7: Response time improvement enabled by independent young generations & regions for short running applications (`javac` and `javap`) executing concurrently with 3 GC-intensive applications – `jess`, `jack` and `ps`. The top graph is for `javac` and the bottom for `javap`. The first bar in each set of bars shows a single instance of the short running program with the GC intensive, long running program, and the second denotes 2 instances of the short program.

short running application. The goal of these experiments is to show the throughput increase (measured as the extra number of serial instances of the small application we can execute), and the response time improvement, of our system versus the shared young generation system. The short applications we consider are `javac` & `javap` with small command-line inputs (which we can provide on request), and the GC-intensive applications are `jess`, `jack` and `ps`.

Bmark	# GCs		Number of tasks									
			1		2		3		4		5	
	Minor	Major	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)	ET (s)	GCT (ms)
jess	146	2	4.59	302	6.17	608	9.58	1001	12.65	1346	16.60	1893
raytrace	76	2	2.82	257	3.76	533	5.74	765	7.25	900	8.98	1157
db	38	2	18.53	255	21.85	638	33.25	1000	43.89	1809	57.16	4164
mpeg	1	1	8.73	50	8.89	95	13.44	149	18.17	190	22.46	272
jack	99	8	4.16	649	5.39	939	7.64	1690	9.38	1706	14.17	2322
ps	217	0	26.67	118	43.96	477	57.67	817	74.59	1272	90.84	1878
jython	142	0	14.32	222	24.75	1408	32.73	2246	42.31	2785	51.22	3446

Figure 4.8: Data for the Base MVM system (shared new generation). Columns 2 & 3 show the number of minor (scavenges) and major collections respectively for a single instance of the benchmark in Column 1. The rest of the columns show execution time (ET) in seconds & GC time (GCT) in milliseconds for 1, 2, 3, 4 and 5 concurrent instances, respectively, of the programs listed. Figures 4.9 and 4.11 show improvement relative to this data.

The results show that in all cases, we enable a significant throughput increase and a response time improvement over a shared young generation system. For `javap`, on average, throughput improvement seems to increase with two concurrent short applications, over a single instance of that application. This is due to the fact that `javap` is very short running and does not exercise GC, and, two instances can be optimally scheduled on our two processor system. For `javac`, throughput gains remain almost the same with two concurrent instances since it does perform stop-the-world GC. Figure 4.7 shows similar trends for the response time. Response time for `javac` is improved by over 15%, while, `javap` shows a 8% to 12% improvement.

In summary, the impact on the execution of a short running program that concurrently executes with another program that shows significantly heap usage, is visibly reduced. This is an effect of performance isolation provided by per-application young generations and fast tenured generation reclamation provided by regions.

We next evaluate the overall performance of our mutator-concurrent scavenging system for a concurrent workload. Figure 4.8 shows data for the original MVM, which is configured with a shared new generation (we henceforth refer to this configuration as the *base*). This includes the number of minor and major GCs, total execution time and GC time for up to 5 concurrent homogeneous instances of the benchmarks.

Figure 4.9 shows the percent improvement in the end-to-end performance enabled by mutator-concurrent scavenging over the base MVM. The mutator-concurrent scavenging configuration includes the old generation regions implementation. The bars represent homogeneous concurrent applications, with one to five applications (left to right bars).

Mutator-concurrent scavenging enables a 10-12% performance improvement for this configuration, across benchmarks on average. `jess` and `jack` show the most improvement (over 20% in many cases), since they involve significantly more GC activity compared to other benchmarks. `raytrace` also shows similar behavior. This is apparent from the data in Figure 4.8.

In the base system all applications must pause for a minor collection triggered by any application, hence applications that cause more GC activity scale poorly. Although `ps` causes a large number of scavenges, the improvement is less pronounced as a percentage of the total execution time due to the fact that the program is long run-

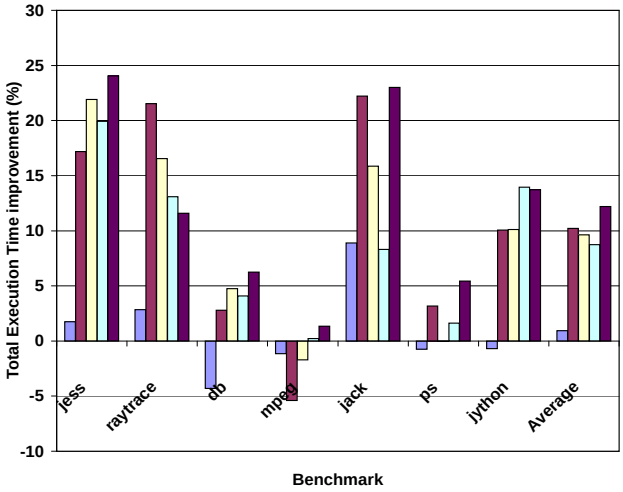


Figure 4.9: Total end-to-end performance improvement enabled by mutator-concurrent scavenging over the base MVM for homogeneous benchmark instances. Bars indicate increasing number of applications (from 1 to 5).

Bmark	Change in # GCs									
	1		2		3		4		5	
	Minor	Major	Minor	Major	Minor	Major	Minor	Major	Minor	Major
jess	9	-2	18	-3	26	-4	34	-4	42	-4
raytrace	5	-1	9	-2	76	-1	95	-2	155	-1
db	2	-1	25	0	57	-1	105	-1	136	-4
mpeg	0	0	0	0	0	0	0	0	0	0
jack	6	-9	11	-9	16	-15	80	-11	26	-11
ps	14	0	25	-1	36	-1	48	-1	58	-2
jython	8	-1	16	-11	23	-15	31	-15	38	-16

Figure 4.10: Change in the number of GCs (minor and major) with mutator-concurrent scavenging over the base MVM for 1 thru 5 instances of the same benchmark.

ning (over a minute). We believe that `mpeg` does not make significant use of the heap and thus, does not reap the benefits from young generation isolation or concurrent allocation techniques. In fact, performance is slightly degraded for this benchmark due to an increase in GC time (explained below).

We next investigate the impact of our techniques on GC activity. Figure 4.10 shows the change in the number of scavenges and full GCs over the base MVM, for

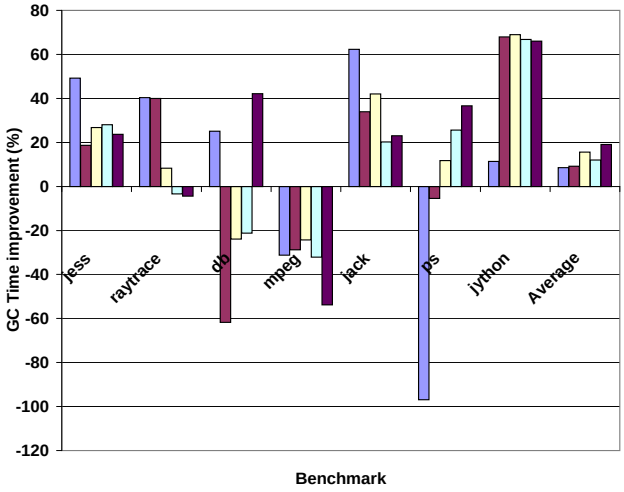


Figure 4.11: Total GC time improvement (minor + major) enabled by mutator-concurrent scavenging over the base MVM. Bars indicate increasing number of homogeneous applications (from 1 to 5).

one to five concurrent homogeneous applications, for each benchmark. We observe that with mutator-concurrent scavenging, the number of scavenges slightly increases in a majority of the programs. The reason for this is that in the base system, a scavenge copies live objects from the entire young generation. Consequently, at the end of the scavenge, the young generation is empty. However, with mutator-concurrent scavenging, promotion is isolated and only the trigger's objects will be promoted. At the end of the scavenge, only one of the young generations will be empty, while the rest may yet trigger a GC since they are allocating independently. However, we perform less work during any single scavenge. Note that `mpeg` shows no change in the number of GCs.

Figure 4.11 shows the percentage change (improvement) in GC time for our implementation versus the base MVM. These figures show a reduction in full GCs with independent scavenging, and a consequent reduction in total GC time, ranging from 9% to 19%. Since, the mutator-concurrent scavenging configuration also includes regions, as applications terminate, other applications start using the terminated applications' freed regions, thus leading to full GC avoidance. Full GCs are much more expensive than scavenges, hence, reduction in full GCs results in a sizeable reduction in overall GC time. Cases in which we are unable to avoid full GCs, do not show an improvement in GC time. In fact, time spent per garbage collection in our system is higher than the base MVM, leading to a performance degradation when the number of GCs is not reduced. This is due to the extra time spent in iterating over a discontinuous set of regions in old generation. This is especially visible in the case of `mpeg`, `db`, and `ps`. Note, however, that GC time is not an indication of overall concurrent system performance.

To summarize, in the base system, every concurrent application will pause on every GC, and therefore experience degraded performance which independent scavenging significantly improves upon. Yet mutator-concurrent scavenging does not impact total GC time adversely although more scavenges are performed than with a shared new generation. Coupled with reclamation of promotion areas, mutator-concurrent

scavenging reduces the number of full GCs, which are generally more expensive than scavenges, resulting in an improvement in total GC time in most cases.

4.2 Discussion

Application-aware GC is thus able to achieve significant performance improvement for concurrent applications, as well as system throughput and scalability when most GC activity is confined to the young generation.

However, old generation collection is performed across all tasks. When applications that make significant use of the old generation trigger an old generation GC, this causes all applications to pause and a global mark-compact GC cycle to execute. Old generation GC across all old generation regions is proportional to the size of the entire old generation. Further, the old generation objects tend to be longer lived. If old generation GC activity constitutes a significant portion of the workload's execution time, performance may suffer. Further, collecting old generation regions that are non contiguous (by design) is challenging – sliding mark-compact GC assumes that the heap memory being collected is contiguous.

In the next section, we motivate the need for and present an old generation GC that combines two different and diverse GC algorithms to achieve high performance,

while maintaining a low footprint for concurrent workloads that make significant use of the old generation.

Figure 4.12 shows the results from a set of experiments that we have conducted to compare MVM [28, 83], with the per-application young generation GC extensions from Section 4.1, with the single-tasking JVM (the Sun Microsystems HotSpot virtual machine version 1.5.0) from which the MVM is derived. The programs are a subset of the benchmarks that we use for our evaluation (that we describe in detail in Section 4.3.2) that exhibit significant garbage collection (GC) activity for the old generation (the longer-lived region). The figure shows that the MVM significantly degrades execution performance for concurrent workloads (2, 5, and 10 concurrent program instances in this graph), despite the significant opportunity for sharing (i.e. multiple versions of the same program are executing concurrently).

The MVM prototype that we use in this study (cf. Section 4.1) achieves performance isolation for the young generation across applications, reclamation of an application's heap memory upon task termination without having to perform GC, per-application accounting of heap usage, and per-application control of heap size settings. However, the extensions described in Section 4.1 still lack complete GC performance isolation, resulting in poor performance versus MVM's single-tasking counterpart for concurrent workloads that fully exercise the memory management system. The key impediment to scalability is the lack of full system GC isolation

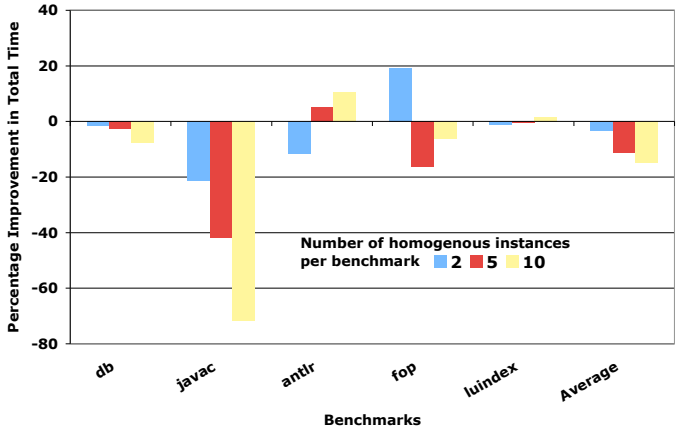


Figure 4.12: Performance of a state-of-the-art multi-tasking MRE (MVM) with per-application young generation GC versus multiple instances of the Java HotSpot virtual machine for *concurrent* execution of five community benchmarks. No prior work has performed such an evaluation. Although per-application young generation GC significantly improves performance over prior state-of-the-art, for programs that involve significant old generation GC activity, performance suffers due to the choice of an unsuitable old generation GC algorithm.

and an unsuitable old generation GC – sliding mark-compact GC that performs compaction over the entire old generation.

To address these issues, we propose a novel *hybrid* GC technique for the old generation that leverages the synchronization mechanism developed earlier (Section 4.1). Our hybrid GC combines two well-known GC algorithms – mark-sweep GC and copying GC in order to achieve high performance, yet, a low memory footprint.

This hybrid GC (i) maintains the constraint that all live objects within a region belong to the same application (which is key to GC isolation and the accuracy of tracking per-application heap usage), (ii) ensures that the aggregate footprint of the multi-tasking MRE is small for concurrent workloads, and (iii) enables space reclaimed

through opportunistic evacuation of objects from sparsely populated regions of one program to be made available to other programs. To achieve these goals, MTM performs full collection of a single program's heap in isolation with co-located concurrent programs by combining fast, space-efficient, mark-sweep collection for regions with little fragmentation, with copying collection for regions with significant garbage and fragmentation.

We have integrated hybrid GC with the MVM prototype described in Section 4.1, and have used it to compare the execution of multiple programs executed using a single multi-tasking MRE versus using multiple concurrent instances of single-tasking MREs (one per program). Two metrics are particularly interesting with respect to the scalability of the two approaches: the overall footprint when executing multiple programs, and the execution times of programs. We demonstrate that with application-aware memory management and hybrid GC, on average, MTM achieves better overall execution times and footprint versus its single-tasking counterpart, for concurrent workloads using a number of community benchmarks. Moreover, MTM is able to do so while maintaining the other benefits of running with a multi-tasking MRE. MTM outperforms the HotSpot single-tasking MRE by up to 14% on average for concurrent instances of the same program (homogeneous), and by up to 16% on average for workloads with a mix of programs (heterogeneous). MTM achieves up to 41% reduction in footprint on average for homogeneous workloads, and by up to 33% on

average for heterogeneous workloads over the single-tasking MRE. Finally, we show that MTM outperforms an extant state-of-the-art multi-tasking MRE by 10% to 22% for concurrent workloads.

In summary, in this section, we describe,

- the first study that compares multi-instance JVM execution versus multi-tasking execution for concurrent program execution;
- a complete memory management system that provides full GC performance isolation for multi-tasking MREs;
- the design and implementation of a hybrid, multi-tasking aware GC that combines GC approaches that are well understood, i.e., mark-sweep and copying, to balance GC performance and memory footprint. Hybrid GC re-uses reclaimed space across multiple isolated program executions; this design achieves footprint-aware memory management that facilitates runtime efficiency for concurrent workloads;
- an empirical evaluation that shows that multi-tasking MREs when equipped with appropriate mechanisms for GC performance isolation, compare favorably to single-tasking MREs with respect to footprint and program execution time for concurrent workloads. This result further strengthens the case for multi-tasking MREs.

4.3 Scalable Hybrid Collection for Multi-Tasking Managed Runtime Environments

MTM's synchronization mechanism allows application threads to be paused on a per-application basis. We leverage this mechanism to design and implement and per-application, old generation GC that is a hybrid of mark-sweep GC and copying GC.

As before, MTM follows the generational design [94] and each application is provided with a private two-generation heap. As with prior versions of MVM, a third generation, called the permanent generation, is shared across applications. The permanent generation is used to allocate long-lived meta-data, such as the runtime representation of classes (including method byte codes, constant pools, etc.), symbols and interned strings, and data structures of the MRE itself, all of which may be transparently shared across programs. The meta-data stored in the permanent generation may survive the execution of many programs, and is rarely collected.

The permanent generation is a single contiguous area. Memory for the young and old generations of applications originates from two pools of fixed-size regions managed by MTM. Each pool uses its own region size. The two pools and the shared permanent generation are contiguous in virtual space, such that old regions are in between the young regions pool and the permanent generation.

As before, memory for the young generation of a program is allocated at program startup, by provisioning a region from the young generation pool. Memory for a program's old generation is allocated on demand, on a per-region basis, from an *old region pool*. Thus, old and young generations are both made of one or more regions, which are possibly disjoint in virtual space. Regions are made of an integral number of operating system virtual pages and aligned to page boundaries to enable on-demand allocation / deallocation of the physical pages allocated to regions by the operating system[1]. Backing storage for the virtual pages of a region is allocated only upon allocation of the region to a program. Conversely, when a region is returned to the pool, the backing storage for its virtual memory pages is freed immediately.

A region can only contain objects allocated by the same program, i.e., a region is always private to a program. This constraint facilitates both tracking of program memory usage and instantaneous, GC-less, reclamation of space upon program-termination [83]. It also helps performance isolation since GC only needs to synchronize with the threads of a single application (instead of all applications).

Tracking of cross-generation references uses a card-marking scheme [21, 50, 45, 13]. Old regions are card-aligned and consist of an integral number of cards, so that young generation collection of an application only needs to scan the dirty cards that correspond to the old regions allocated to the application. These are maintained

[1]E.g., using `map`/`unmap` system calls on the Solaris^{TM} OS.

in a per-application list ordered by increasing virtual address. Each application is also associated with a *current* old region, which identifies the region used to allocate tenured space for the applications. Tenured space is allocated primarily during young generation collection, when promoting young objects, and occasionally, directly by mutator threads of the application to allocate space for large objects.

MTM initiates a young generation collection for an application when the application's young generation is full, and an old generation collection when the application reaches its maximum heap size limit, or when allocation of a region from the pool of old region fails. Minor collection for an application is performed concurrently with respect to other applications using mechanisms described previously [83].

Collection of the old generation of an application's heap space follows a *hybrid* approach that combines fast, space-efficient, mark-sweep for regions of the old generation with little fragmentation or garbage, with a copying collection for regions of the old generation with either significant fragmentation or with a significant amount of garbage. Old generation collection is on a per-application basis, i.e., only the old generation of the application that triggers GC is collected.

We also exploit MVM's representation of classes to organize the permanent generation in a way to limit tracing, during young and old generation collection, to objects of the application that initiated the collection (henceforth called the *GC initiator*). The MVM separates the application-dependent part of the runtime represen-

tation of classes from the rest of the class representation. When a class is sharable across applications, a *task table* is interposed between the class representation and its application-dependent part, the latter being allocated in the old generation of the corresponding application. The task table for a class has an entry for every application executing in the MRE, and each application is assigned, upon startup, a unique number (*the task identifier*) that is used to index these tables. The entry of a task table holds a reference to the object that holds the application-dependent part of the class when the application associated with that entry loads the class, or a null pointer otherwise [28]. Classes whose representation cannot be shared across programs (e.g., classes defined by program-defined class loaders) refer directly to the application-dependent part. All data structures that directly reference application-dependent data are clustered in a specific area of the permanent generation, which is the only area that must be traced during collection of younger generations. When an application does not use program-defined class loaders, tracing is limited to a single entry in every task table (the entry assigned to the GC trigger).

Other data-structures that reference application-dependent data (e.g., JNI Handles) are organized either in a per-application pool or in tables with one entry per application, similar to the task table. We exploit this to *scan only those pools or table entries associated with the GC initiator*. Further, only stacks of threads of the GC initiator are scanned for roots.

We describe MTM's hybrid garbage collector in detail in the next section.

4.3.1 Hybrid Mark-Evacuate-Sweep Garbage Collector

Our experiments with prototypes of MVM suggest that efficient GC is key to making the concurrent execution of multiple programs using multi-tasking a viable alternative to running the same programs using one instance of a single-task MRE per application.

MTM's old generation design is constrained by the need to ensure that an old region contains only objects from the same application, for performance isolation, as well as for efficient and accurate tracking of heap resources. This implies that dead space within an old region allocated to an application cannot be reused by another application. This can potentially lead to significant fragmentation and substantially increase footprint for multi-tasking. Copying GC is effective at mitigating fragmentation, but at the cost of excessive copying of live objects, and the necessity of a copy reserve area. In place compaction requires multiple passes over the heap (although recent work has significantly optimized compaction [64]). Mark-sweep, however, is fast, and involves a single pass over live data, but may result in poor space utilization [61].

MTM combines two relatively simple and well-understood techniques: mark-sweep and copying. We use copying to evacuate live objects from only those regions

that are fragmented or are sparsely populated, and mark-sweep for the remaining regions. The goal is to maintain a low footprint, but without the overhead of copying of all live objects and a copy reserve for every GC. Space reclaimed via sweeping can only be used by the GC initiator, since the free space may be co-located with live data in the same region. Evacuated regions, on the other hand, can be returned to the old region pool where the backing storage for their virtual pages is freed until the regions are re-assigned to an application.

Candidate regions for evacuation are selected based on the amount of fragmentation the GC *is likely to cause* in the region. Before the collection begins, MTM suspends all the threads of the GC initiator at a GC *safepoint*. The threads are restarted when GC completes.

The collection itself is performed in four phases: marking, selecting candidate regions for evacuation, evacuation (copying), and sweeping and adjustment of regions (performed in the same pass). The first two phases gather information (liveness, connectivity, occupancy, and estimated fragmentation) necessary for the last two phases. Evacuation and adjustment are optional, and occurs only if the second phase selects regions for evacuation.

Figure 4.13 illustrates with an example the main phases of MTM's hybrid collection. The following sub-sections detail each of the four phases.

Marking Phase

The marking phase begins a collection and produces two data structures as output: a *mark bitmap* that records live objects of the GC initiator; and a *connectivity matrix* that records connectivity information between old generation regions. Together, these are used to determine regions to evacuate, sweep and adjust.

Storage for the mark bitmap and the connectivity matrix is allocated for the duration of the hybrid GC cycle. The mark bitmap has one bit for every word of heap memory. Marking starts with the roots of live objects for the GC initiator: the stacks of the application's threads; the entry corresponding to the GC initiator in each task table for the runtime representation of shared classes in the permanent generation, and entries in global tables maintained by the multi-tasking MRE (such as JNI handles).

Marking then traverses the object graph from these roots. Because isolation is strictly enforced between applications through application-private regions, the marking phase will never access an object allocated by another application nor traverse a region allocated to another application.

The connectivity matrix is updated when a yet unmarked object is traversed. The matrix is encoded as a two-dimension boolean array, so that an entry (i, j) set to *true* indicates that there exists *at least* one reference from region i to region j. The matrix is initially zero-filled.

Testing whether each reference crosses a region boundary can be expensive. We have observed that inter region object references in the old generation are clustered, and that the distance between the referencing (source) object and the object being referenced (destination) in an old region is often small. Therefore, given an old region size that is large enough, the source and destination objects are likely to reside in the same region. If region size is a power of two, and regions are aligned, testing whether two addresses are in the same region can be inexpensively performed as follows [2]:

```
to == *from;
if (to ∧ from) >> LOG_REGION_SIZE) != 0 {
   // Not in the same region.
   update_connectivity_matrix(to,from);
}
```

When the test fails, a slow path is taken in order to update the connectivity matrix. The choice of an appropriate region size contributes to confine clusters of connected objects to a single region, which has two benefits: reducing the overhead of tracking inter-region connectivity (i.e., the fast path will be taken more often); and limiting the number of regions that must be inspected for potential pointer adjustment after

[2]This test for cross-region object references is similar to the test in the write barrier of the Beltway framework [15] except that, in our case, the test is performed at marking-time.

regions are evacuated. We have empirically identified an old region size of 256KB that works well.

In summary, tracking of connectivity information helps to reduce the amount of live data that must be scanned during pointer adjustment if any region is evacuated.

Selecting Regions for Evacuation

The decision to evacuate a region attempts to balance the cost of evacuation (copying and pointer adjustment) and heap fragmentation (consequently, footprint). To maintain a low cost for evacuation we evacuate sparsely populated regions, while to maintain a low footprint, we evacuate regions with fragmentation.

That is, our evacuation policy evacuates a region unconditionally if the ratio of the live to dead space (*live ratio*) is less than a certain minimum live ratio ($MinLiveRatio$). The region is also evacuated if it is estimated to be fragmented. This is done by comparing the average size of each contiguous fragment of dead space to a threshold ($MinFragmentSize$). That is, given L, the amount of live data in the region, F, the number of contiguous areas of dead objects in the region, and R the size of the region, a region is selected for evacuation if:

$$(L/R) < MinLiveRatio \vee ((L/R) < K \wedge (F > 1) \wedge ((R-L)/F) < MinFragmentSize)$$

We empirically chose $MinLiveRatio$ to be 0.25, i.e., a region is always evacuated if it contains over 75% of garbage. When the pool of old regions is closed to

exhaustion, this parameter is increased up to 0.9 to aggressively evacuate all but the almost full regions. K is the occupancy threshold and chosen to be 0.9, i.e., we look for fragmentation in regions that are at least 90% full. $MinFragmentSize$ is set to 50 bytes by default.

In order to realize this policy, MTM needs to determine the ratio of live to dead data and the number of contiguous fragments of dead space in each region. This is done following the completion of the marking phase, by scanning the mark bitmap. For each region belonging to the GC initiator, MTM walks over the region's objects, using the mark bitmap to determine their liveness and the objects' header to obtain their size. In addition, in this pass, adjacent dead objects are coalesced into a single dead area to reduce scanning time for future passes.

Evacuation, Sweeping and Adjustment of Old Regions

Live objects in regions selected for evacuation are relocated to new regions allocated from the old regions pool. Evacuation traverses the region being evacuated for live objects using the mark bitmap. Live objects are copied to the new region, and a forwarding pointer is installed in the header of the (old) copied object. Forwarding pointers are necessary for pointer adjustment. This, however, prevents the evacuated regions from being freed before adjustment is complete.

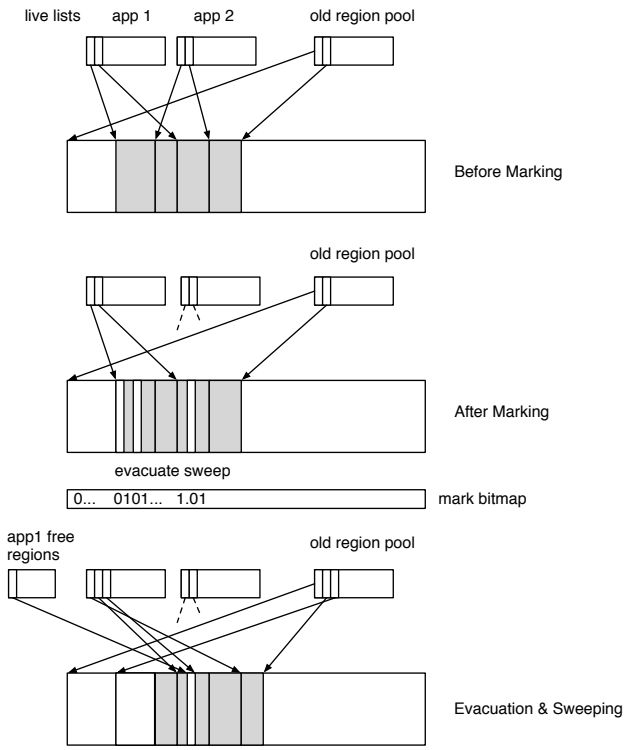


Figure 4.13: Marking, Evacuation and Sweeping of Old Regions. Each application has a corresponding list of live areas. Marking traverses live objects for an application and marks live objects in the mark bitmap. After marking, candidate regions for evacuation (or sweeping) are selected based on the amount of live data and fragmentation. Regions selected for evacuation are then evacuated, regions selected for sweeping are swept and free areas in these added to a per-application free list. Pointer adjustment for swept regions is also performed during this pass, if necessary.

New regions used to store evacuated objects are added to the set of regions that need adjustment, i.e., we assume that the a region is likely to contain objects that point to other objects in the same region.

Once evacuation is complete, sweeping and adjustment of pointers can be performed in the same pass. For each region that was not evacuated, the mark bitmap

corresponding to the region is used to build lists of live and free areas within the region. The connectivity matrix is also checked to determine if the region contains objects that reference objects in evacuated regions. If so, the live objects in the region are scanned to adjust references. Finally, the free and live lists of areas are combined into a list of live old generation areas used by the application. The live list is used to account for the old generation usage of the application, as well as during young generation collection to limit the amount of work that is done during card scanning, i.e., we only need to scan dirty cards that correspond to the application's list of old generation regions. The application's free list that was constructed during sweeping can only be used to satisfy allocation requests for that application.

If any region is evacuated, in addition to adjustment of some old regions, we also need to adjust objects in the young generation of the application, the permanent generation, and outside the heap (globals) that reference objects in the evacuated region(s).

The young generation is typically small (the default is 2MB) and can therefore be scanned in its entirety without significant overhead. However, performing an object graph traversal beginning from the roots to identify globals and permanent generation pointers that must be adjusted can be prohibitively expensive. Instead, we keep track of the locations of these pointers during the marking phase, and update them during

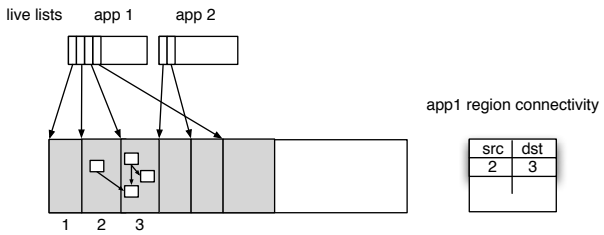


Figure 4.14: Adjustment of old regions. Application 1 is being collected. We build the region connectivity matrix for application 1 during the marking phase. Region 2 has outgoing pointers to Region 3, therefore, Region 2 must be scanned if Region 3 is evacuated. However, Region 1 and 4 do not must be scanned.

pointer adjustment. The space overhead required to keep track of these locations is small, and is reported as part of the total footprint of MTM in Section 3.5.

Once all regions have been adjusted, the evacuated regions are returned to the pool of free regions, and backing physical memory corresponding to their virtual address pages is unmapped, i.e., freed regions do not consume physical memory and can be later re-mapped and used as part of the old generations for *any* application.

Objects larger than a single region are treated specially. They are assigned an integral number of contiguous old regions large enough to hold the object. MTM notes whether a region is part of a single large object region and whether that object contains no references (scalars only). This information is used to reclaim space when these large objects die (e.g., by returning their regions immediately to the pool, without waiting for adjustment).

4.3.2 Evaluation

The design of MTM was motivated partly by the poor behavior of extant approaches to multi-tasking for concurrent application workloads.

In this section, we report our assessment of how well a multi-tasking MRE using MTM fares with concurrent workloads. We first compare the performance of MVM with per-application young generation GC (the extensions described in Section 4.1) to MVM modified to integrate hybrid old generation GC as well. We then compare MTM to a single-tasking MRE. We use the Java^{TM} HotSpot virtual machine, version 1.5.0-03 [91], a production quality, high-performance virtual machine from Sun Microsystems (which we will refer to as HSVM from now on). MTM derives its implementation from HSVM and shares a significant amount of code, which facilitates a fair comparison. MTM differs only in its memory management sub-systems and modification to the runtime to achieve GC performance isolation. All other mechanisms to support multi-tasking and sharing of the runtime representation of classes, byte code and compiled code (see [28, 29] for detailed descriptions) as well as other virtual machine implementation aspects inherited from HSVM are identical.

The main metrics of interest for our comparison are execution time and the aggregate memory footprint necessary to run concurrent workloads. We begin with a description of our experimental setup, including hardware, benchmark, and methodology.

Benchmark	Description
compress	SpecJVM98 compression utility (input 100)
jess	SpecJVM98 expert system shell benchmark: Computes solutions to rule based puzzles (input 100)
db	SpecJVM98 database access program (input 100)
javac	SpecJVM98 Java to bytecode compiler (input 100)
mtrt	SpecJVM98 multi-threaded ray tracing implementation (input 100)
jack	SpecJVM98 Java parser generator based on the Purdue Compiler
antlr	Dacapo antlr: Parses grammar files and generates a parser
	and lexical analyzer for each (default input)
fop	Dacapo fop: XSL-FO to PDF parser/converter (default input)
luindex	Dacapo luindex: uses lucene to index the works of Shakespeare and the King James Bible (default input)
ps	Dacapo ps: Postscript interpreter (default input)
opengrok	Open source code browser and cross reference tool (input: Source files in HSVM "memory" subdirectory, 118 files)
jruby	Ruby interpreter written in Java (uses small scripts as input: beer song, fibonacci numbers, number parsing, thread test)
groovy	Groovy interpreter written in Java (input: unsigns, i.e., strips MANIFESTs) for a number of jar files from Apache ant)
antlr-mixed	mixed workload consisting of antlr, fop and opengrok
luindex-mixed	mixed workload consisting of luindex, fop and ps
javac-mixed	mixed workload consisting of javac, jess, mtrt and jack
scripts-mixed	mixed workload consisting of groovy and jruby

Figure 4.15: Benchmarks and workloads used in the empirical evaluation of MTM

Experimental Methodology

We ran our experiments on a dedicated dual CPU 1.5GHz UltraSPARC IIIi system, with 2GB physical memory, 32KB instruction and 64KB data cache running the Solaris^{TM} Operating System version 10. Figure 4.15 describes the benchmarks and workloads we used for our experiments.

Programs used in our concurrent workloads are selected from community programs from the SpecJVM98 [85] and Dacapo [31] benchmark suites [3], as well as two commonly used open-source scripting applications, `jruby` [62] and `groovy` [41], and an open-source source code browser and cross reference tool called `opengrok` [76].

[3] We used version 2006-10-MR2 version of the Dacapo benchmarks, and `ps` from Dacapo version beta-05022004.

	Number of instances					
	2		5		10	
Bmark	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)
compress	10.96	139.44	27.60	351.16	56.41	650.80
jess	4.93	19.82	12.33	33.18	24.54	55.12
db	20.84	35.95	52.50	74.05	105.10	141.00
javac	10.40	49.51	26.78	109.85	53.97	261.03
mtrt	3.39	20.46	8.47	62.27	16.24	114.26
jack	4.21	30.84	10.75	59.53	20.89	104.78
antlr	9.17	67.51	20.95	114.23	39.86	219.61
fop	6.00	51.84	14.31	87.45	28.53	148.49
luindex	40.08	76.68	89.45	173.35	169.42	333.43
ps	27.18	16.63	68.37	23.91	136.80	37.02
opengrok	10.44	101.50	25.40	230.85	51.35	429.77
groovy	10.15	138.06	21.54	366.63	50.92	544.25
jruby	2.58	34.80	5.66	49.67	10.67	73.47
Average	12.33	60.23	29.55	133.55	58.82	239.46

	Number of instances			
	1		2	
Bmark	Exec time (sec)	Footprint (MB)	Exec time (sec)	Footprint (MB)
antlr-mixed	12.64	79.52	24.43	148.06
luindex-mixed	34.44	77.04	42.47	132.76
javac-mixed	13.28	31.97	23.58	63.57
scripts-mixed	8.28	68.68	11.30	118.95
Average	17.16	64.30	25.45	115.84

Figure 4.16: Total execution time (in seconds) and footprint (in MB) data for MTM with application-aware memory management and hybrid old generation GC for concurrent homogeneous (multiple instances of same application), and heterogeneous (multiple instances of different applications). The benchmarks are described in Figure 4.15. All relative performance improvement results for execution time as well footprint in this section use these values.

We exclude *mpegaudio* from SpecJVM98 (as is commonly done) since it does not exercise the GC.

We experimented with two types of workloads: *homogeneous* and *heterogeneous*. A homogeneous workload consists of multiple concurrent instances of the same program. For instance, "10 instances of `javac`" implies that 10 instances of this program are launched simultaneously. A heterogeneous workload consists of concurrent instances of different programs.

We refer to the heterogeneous workload as *mixed* in Figure 4.15. We present results for 1 and 2 instances each of an application in a heterogeneous workload. For

example, 2 instances each for `antlr-mixed` implies that we launch 2 instances each of `antlr`, `fop` and `opengrok` simultaneously, i.e., 6 concurrent instances.

We report total execution time by reporting the time elapsed since the applications in a workload were launched and until the last application completes. We use a harness that executes each application as an *isolate* [57] using reflection and we report total elapsed time using `System.nanoTime()`. To measure footprint, we use the UNIX `pmap` utility, which we execute as an external process in a tight loop and report the maximum of the total RSS (resident segment size) value reported by executing `pmap -x` on the MRE process. Footprint and execution times are reported using independent runs. In case of single-tasking, we sum the RSS values returned by `pmap` for each individual MRE process (since to execute concurrent workloads, we must launch a single-tasking MRE process for each application).

We perform all HSVM experiments using the *client* compiler and the default serial GC (sliding mark-compact) used for client configuration (i.e., using the `-client -XX:+UseSerialGC` command line flags). HSVM and MTM both use copying GC for collecting the young generation. For all results, we present the average of 5 executions.

MTM with Per-Application Hybrid GC Versus MVM

We first present performance results for the improvement enabled by MTM with per-application hybrid GC versus the prior MVM version that provides per-

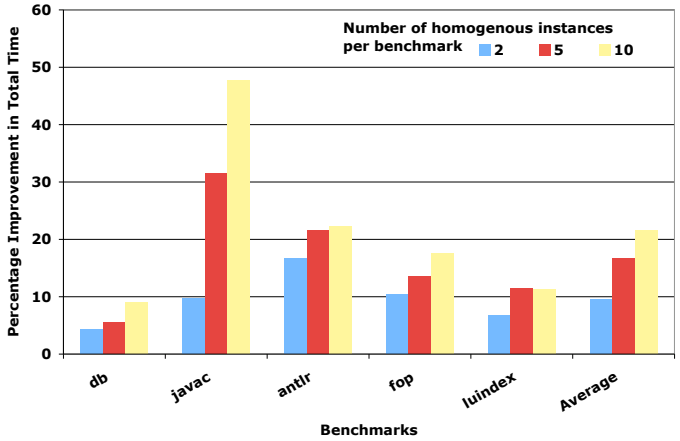


Figure 4.17: Percentage improvement in execution time enabled by MTM (MVM extended with per-application hybrid GC) with per-application hybrid GC versus a prior implementation of MVM (cf. Section 4.1) when executing concurrent workloads that show significant old generation GC activity. MTM enables better performance due to a more efficient old generation GC and performance isolation.

application young generation GC, but a mark-compact old generation GC that collects the entire old generation heap (i.e., for all applications) (cf. Section 4.1). As seen earlier, this prior version provides performance isolation for the young generation, yet performs poorly for concurrent workloads relative to executing the same concurrent workload with multiple instances of HSVM (cf. Figure 4.12) for applications that show significant old generation usage.

Figure 4.17 shows the performance improvement enabled by MTM over this MVM. The results indicate that MTM outperforms MVM by 10%, 15%, and 22% on average when running 2, 5, and 10 concurrent instances, respectively. For this experiment, we only present results for applications that show significant old generation GC activity. This performance improvement is possible due to the hybrid old

	Number of instances								
	2			5			10		
Bmark	MVM (sec)	MTM^2 (sec)	% imp	MVM (sec)	MTM^2 (sec)	% imp	MVM (sec)	MTM^2 (sec)	% imp
db	0.57	0.28	**51.95**	2.92	0.70	**76.05**	5.47	1.38	**74.81**
javac	3.24	2.51	**22.47**	8.95	3.48	**61.06**	40.10	7.93	**80.23**
antlr	2.44	0.48	**80.17**	4.11	1.29	**68.69**	6.23	1.39	**77.75**
fop	1.18	0.67	**42.96**	2.29	1.11	**51.58**	4.98	2.54	**49.00**
luindex	4.27	1.51	**64.73**	8.36	2.86	**65.82**	14.35	8.24	**42.60**
Average	2.34	1.09	**52.46**	5.32	1.89	**64.64**	14.22	4.29	**64.88**

Figure 4.18: Old generation GC times (total) for MTM (MVM extended with per-application hybrid GC) versus a prior implementation of MVM described in Section 4.1. GC times are presented in seconds along with percentage improvement in GC time enabled by MTM . MTM 's per-application hybrid old generation GC outperforms mark-compact old generation GC used in the prior implementation.

generation GC in MTM that enables performance isolation, as well as improved old generation GC performance.

Figure 4.18 shows the old generation GC times for MTM versus the prior version of MVM. MTM's hybrid GC significantly improves GC performance. The prior MVM version uses a stop-the-world mark-compact GC for the old generation that performs three passes over the entire old generation (for all applications), with cost proportional to the size of the heap. With more concurrent instances, the cost of old generation GC increases.

In addition, MTM never pauses tasks to perform GC and all allocation and collection for any application is isolated with respect to other applications. MTM scales better overall due to performance isolation as the number of instances is increased, as seen in Figure 4.17. The impact of performance isolation is especially evident in the case of `javac`. For instance, when 10 concurrent instances of `javac` exe-

cute, the total old generation GC time with full heap mark-compact GC is about 40 seconds. The cost of old generation GC is higher since mark-compact GC needs to scan a larger heap. Further, since *all* applications are paused during old generation GC, performance for *javac* is significantly degraded. In the case of *db* and *luindex*, GC time does not dominate total execution time, and consequently, the improvement enabled by MTM's hybrid GC is less significant.

MTM with Per-Application Hybrid GC Versus HSVM

We next compare the execution time and footprint of MTM with per-application hybrid GC to HSVM. HSVM allows users to specify an initial heap size (32MB by default) and a maximum heap size (64MB by default) when launching an application. The initial heap size controls the heap limit, the point at which a full GC is triggered. The initial heap size grows (or shrinks) after a full GC, if required. For results in Figures 4.19, 4.20, 4.21, and 4.22, we set the initial heap size for HSVM equal to the maximum heap size. With this setting, HSVM performs less frequent GC and achieves better overall performance (total execution time), compared to when the initial heap size is at the default value. This setting allows single-tasking to perform at its best potential since the application heap is not restricted. We also present results for the other case, i.e., when the initial heap size for HSVM is not set to the maximum initially (the default behavior), thereby allowing HSVM to achieve a smaller footprint (Figures 4.25, 4.26, 4.23 and 4.24). MTM does not restrict the initial heap size, or

use a "soft limit" for applications, yet *we always ensure that we never exceed the maximum heap size setting for an application* (which is set to the HSVM default maximum heap size of 64MB in order to ensure a fair comparison).

Figure 4.19 shows percentage improvement in total execution time when homogeneous workloads are executed with MTM versus the HSVM virtual machine, i.e., concurrent instances of the same application. We present results for 2, 5 and 10 concurrent instances for each application. Multi-tasking allows sharing of compiled code and classes between applications resulting in reduced overall execution time. MTM enables an improvement of 11%, 13% and 14% for 2, 5 and 10 concurrent applications on average for homogeneous workloads. MTM allows complete application isolation and space reclaimed by evacuating old generation regions for an application to be reused by other applications. Scripting and parsing applications such as `antlr` and `jruby` are commonly used on desktop systems and particularly show a significant benefit due to sharing of compiled code.

For some applications, such as `compress`, `javac` and `ps` multi-tasking does not outperform single-tasking. For `compress` in particular, multi-tasking performance lags single tasking due to the fact that it allocate large objects (byte arrays) in the old generation which leads to fragmentation and worse GC performance in a shared old generation address space, and also due to the overhead due to a level of indirection to access static variables [28]. However, MTM attempts to mitigate the ad-

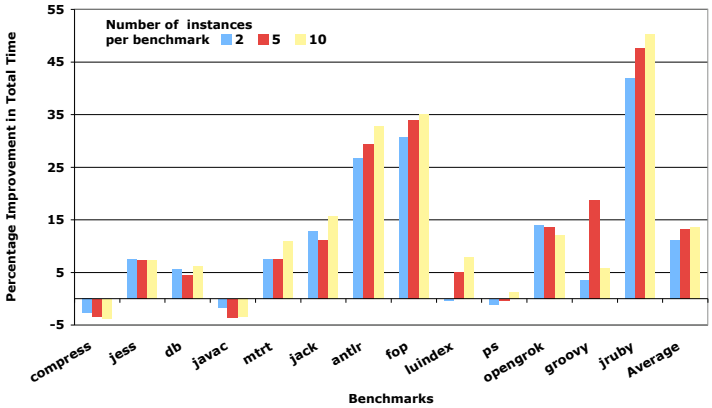


Figure 4.19: Percentage improvement in execution time enabled by MTM over HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (multiple instances of the same application). Benchmarks are described in Figure 4.15.

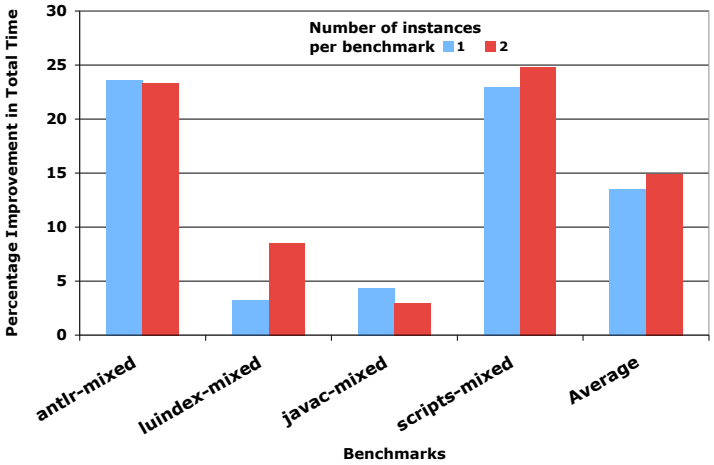


Figure 4.20: Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.15.

verse impact of fragmentation and achieves a significant benefit for these worst-case applications over the state-of-the-art multi-tasking MRE implementation, as shown earlier (cf. Figure 4.17), while achieving performance that is close to the perfor-

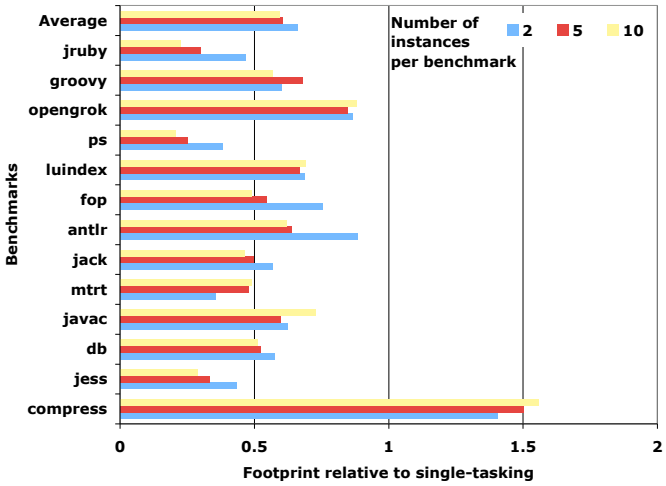


Figure 4.21: Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application).

mance of these applications with single-tasking (within 3%). On average, MTM significantly outperforms single-tasking.

Figure 4.20 shows the percentage improvement in total execution time for heterogeneous workloads, i.e., concurrent instances of different applications for 1 instances of each application, and 2 instances of each application for every heterogeneous workload (see Figure 4.15). For example, *antlr-mixed* with two instances indicates that 2 instances each of antlr, fop, opengrok are executed concurrently (6 concurrent applications). On average, MTM improves performance by up to 16%, with improvements ranging from 3% to 25% in individual cases. As seen earlier, scripting workloads in particular perform very well with multi-tasking.

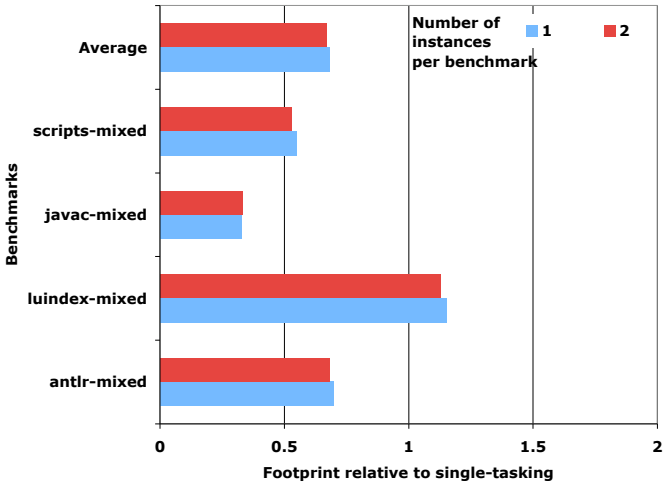


Figure 4.22: Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = max heap size = 64MB) for heterogeneous concurrent workloads. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix.

Figures 4.21 and 4.22 compare the total process footprint for MTM versus HSVM for the same set of applications as in the previous figures. Each bar represents the ratio of the footprint for MTM versus HSVM. The value 1 indicates that MTM and HSVM have identical footprint for a given workload. Values less than 1 indicate that MTM has a lower footprint.

MTM shows a better footprint compared to HSVM and on average, MTM achieves 34% to 41% reduction in footprint for homogeneous workloads, and 31% to 33% benefit for heterogeneous workloads. These savings are possible due to sharing of classes and compiled code in a multi-tasking MRE.

However, `compress` shows worse footprint (around 50% or 1.5x). The worse footprint for `compress` can be attributed to large scalar objects (objects that do not

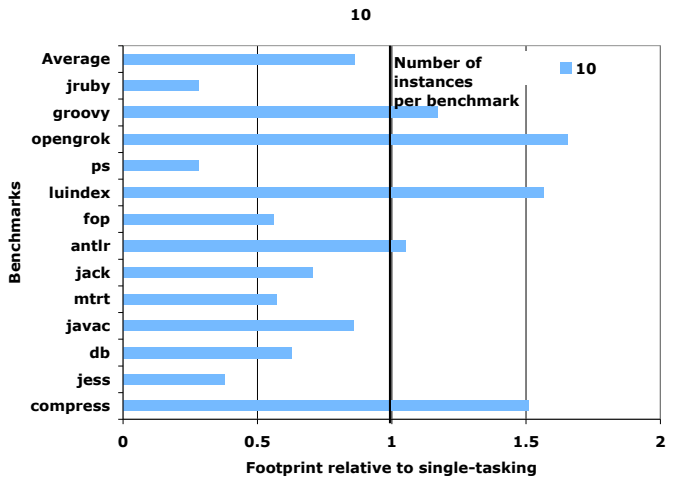


Figure 4.23: Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = 32MB) for homogeneous concurrent workloads (2, 5, and 10 instances of the same application).

hold references, such as byte arrays). As noted earlier, `compress` allocates a significant number of large byte arrays, which are directly allocated in the old generation. Since our old generation is non-contiguous, and since we allocate large scalar objects in a separate region, which we can safely skip during pointer adjustment, allocation of very large ($>$ minimum region size, which is 256KB by default), byte arrays leads to excess fragmentation. A new region must be allocated for each such large byte array, and this region must be aligned to the region boundary for correctness. However, the number and size of these is unknown at runtime, without a priori profiling. Therefore, we cannot pre-allocate a suitable sized region. As part of future work, we plan to address the allocation of large objects, by providing a per-application large object region that is sized differently and collected separately from the old generation.

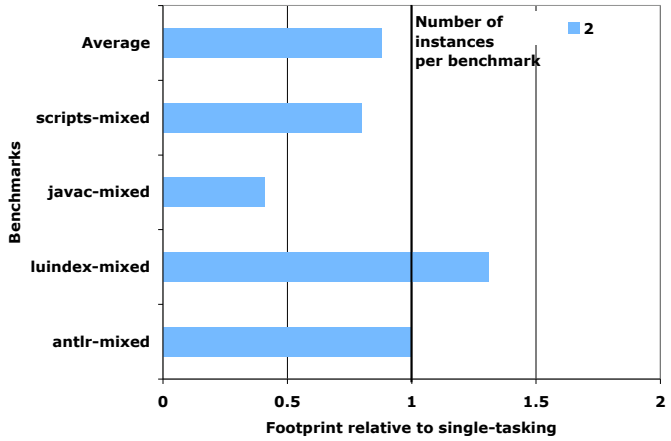


Figure 4.24: Percentage improvement in footprint enabled by MTM versus HSVM (default initial heap size = 32MB), heterogeneous workloads, i.e., multiple concurrent instances of different applications. 1 denotes 1 instance each of the mix of applications that constitute a heterogeneous workload. 2 indicates 2 instances of each application in the mix.

Note that `compress` is a numerical computation benchmark and does not represent typical MRE workloads.

Figures 4.23 and 4.24 compare the process footprint for MTM versus HSVM, when the initial heap size for HSVM is restricted and increased gradually. In this configuration, HSVM gradually increases the heap (if required), starting from an initial default (32MB), in order to achieve smaller footprint. As expected, HSVM runs in a much smaller heap and consequently, the process footprint is lower. On average, MTM shows a footprint improvement of 6% to 14% for homogeneous workloads, and 12% to around 15% for heterogeneous workloads. Note that these values are smaller compared to the earlier configuration of HSVM, i.e. when the initial heap size for HSVM is not restricted. However, if we look at the execution time for MTM

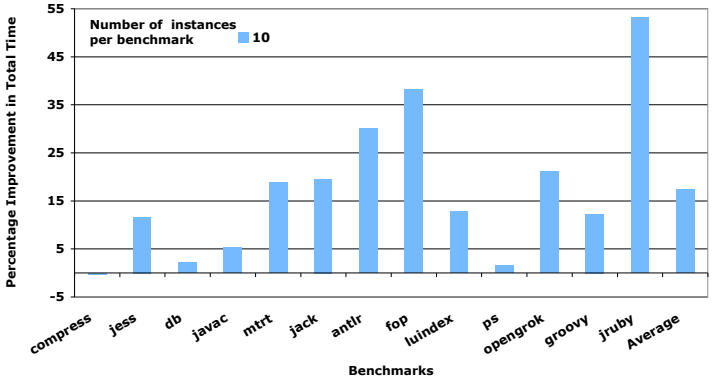


Figure 4.25: Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = 32MB) homogeneous concurrent workloads. Benchmarks are described in Figure 4.15.

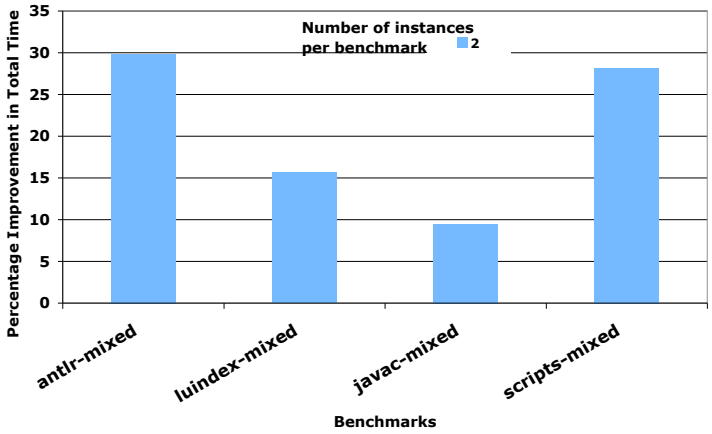


Figure 4.26: Percentage improvement in execution time enabled by MTM versus HSVM (default initial heap size = 32MB) for heterogeneous concurrent workloads (multiple instances of different applications). Benchmarks are described in Figure 4.15.

versus HSVM (Figures 4.25 and 4.26) when the initial heap size for HSVM is restricted, MTM outperforms HSVM by a *greater margin* than when we do not restrict the initial heap size for HSVM. On average, MTM shows an improvement of 15% to over 17% for homogeneous workloads, and 19% to 21% for heterogeneous workloads.

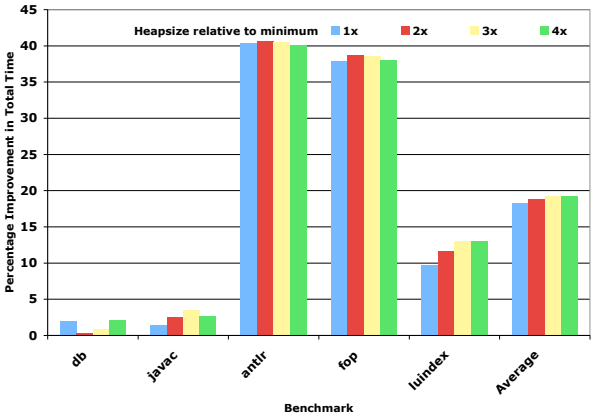


Figure 4.27: Percentage improvement in execution time enabled by MTM over HSVM for 1 through 4 times the minimum heap size that each benchmark needs to execute in MTM .

In summary, by controlling heap growth the single-tasking HSVM virtual machine can achieve a better footprint when the heap is not restricted, however, MTM shows a comparable or better footprint on average across concurrent workloads that we looked at. Further, MTM outperforms HSVM by a larger margin, since there is a reduction in performance for the single-tasking MRE due to more frequent GC. There exists a tradeoff between execution time and footprint by choosing the threshold at which GC is triggered. We believe that manually having to select an appropriate per-application heap size in a context of a multi-tasking MRE is counter-productive. On average, MTM significantly outperforms HSVM and has a better footprint without having to manually select an appropriate initial per-application heap size.

We next examine the performance of MTM versus HSVM as the heapsize is varied from the minimum that an application requires to execute in MTM , to 4

times the minimum for that application (Figure 4.27). We only consider benchmarks that show significant old generation GC activity. The minimum heap size selected is 45MB for `luindex` and 22MB for the rest. Across heap sizes, MTM outperforms HSVM by 18 – 19% on average.

However, HSVM is able to execute programs in a smaller heap compared to MTM (i.e., $<$ 45MB for `luindex` and $<$ 22MB for other benchmarks). HSVM uses in-place sliding compacting GC, which is more space efficient than MTM's hybrid GC for small heaps. This is due to the fact that evacuation, although it is partial and selective, requires a copy reserve for the duration of the GC to copy live objects. For highly memory constrained scenarios, HSVM's GC may be a more suitable choice compared to evacuation. We are investigating mechanisms to perform in-place compaction across disjoint regions as part of future work.

Sensitivity Analysis

In the next set of results, we examine how MTM with selective evacuation (copying) and mark-sweep compares to only mark-sweep and only copying. Our hybrid GC can operate as a mark-sweep only GC (by setting the $MinLiveRatio$ threshold described in Section 4.3 to `0`), or as a copying only GC (by setting the $MinLiveRatio$ threshold to `1`, i.e., 100%).

In particular, in Figure 4.28 we present total process footprint for MTM with hybrid GC versus MTM with mark-sweep only, and MTM with copying only, for

	Number of instances														
	2					5					10				
				% imp vs					% imp vs					% imp vs	
Bmark	MTM^2 (KB)	MTM^2 MS (KB)	MTM^2 CP (KB)	MS	CP	MTM^2 (KB)	MTM^2 MS (KB)	MTM^2 CP (KB)	MS	CP	MTM^2 (KB)	MTM^2 MS (KB)	MTM^2 CP (KB)	MS	CP
javac	49.5	127.7	65.4	**61.2**	**24.3**	109.9	297.5	117.7	**63.1**	**6.6**	261.0	602.1	261.7	**56.6**	**0.3**
luindex	76.7	128.4	83.5	**40.3**	**8.2**	173.4	302.9	182.0	**42.8**	**4.7**	333.4	589.5	351.9	**43.4**	**5.2**

	Number of instances									
	1					2				
				% imp vs					% imp vs	
Bmark	MTM^2 (KB)	MTM^2 MS (KB)	MTM^2 CP (KB)	MS	CP	MTM^2 (KB)	MTM^2 MS (KB)	MTM^2 CP (KB)	MS	CP
antlr-mixed	79.5	86.1	80.6	**7.6**	**1.3**	148.1	156.6	148.2	**5.5**	**0.1**
javac-mixed	32.0	51.9	41.6	**38.4**	**23.2**	63.6	91.3	87.5	**30.4**	**27.4**
scripts-mixed	68.7	94.8	104.5	**27.5**	**34.3**	119.0	127.0	140.3	**6.4**	**15.2**

Figure 4.28: Footprint for MTM with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for a set of homogeneous (instances of the same application) and heterogeneous (different applications) concurrent workloads. Hybrid GC achieves a footprint that is lower than always choosing mark-sweep or always choosing copying.

a subset of benchmark programs. We only present results for benchmarks that show significant change in footprint compared to either mark-sweep or copying ($> 5\%$). For all other benchmarks, we did not find a significant change in the footprint (however, MTM with hybrid GC never shows a worse footprint compared to either mark-sweep or copying).

For `javac`, `luindex`, `javac-mixed` and `scripts-mixed`, hybrid GC has a much smaller footprint compared to mark-sweep. We believe this is due to fragmentation due to using mark-sweep only without any compaction. For `javac-mixed` and `scripts-mixed`, copying has a higher footprint, since always copying all live data requires a larger copy reserve space during GC. While performing evacuation,

	Number of instances														
	2					5					10				
				% imp vs					% imp vs					% imp vs	
Bmark	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP	MTM^2 (sec)	MTM^2 MS (sec)	MTM^2 CP (sec)	MS	CP
javac	10.40	10.82	10.57	**3.9**	**1.6**	26.78	28.14	27.29	**4.8**	**1.9**	53.97	56.48	55.25	**4.4**	**2.3**

Figure 4.29: Execution time MTM with hybrid GC (mix of mark-sweep and copying) versus mark-sweep (MS) only and copying GC (CP) only for the `javac` benchmark.

the old as well as the new (copied to) regions must be occupied (mapped) for the duration of the GC cycle.

We next examine the effect of using hybrid GC, mark-sweep only, and copying only, on execution time for `javac`, which shows a significant difference in performance (Figure 4.29). Using mark-sweep only results in excess fragmentation. Fragmentation has an interesting effect on execution time for `javac` – an increase in young generation GC time by 8% on average (or 0.51 sec, 0.66 sec and 1.17 sec for 2, 5 and 10 instances respectively) due to an increase in card scanning time, since more cards must be scanned. Using copying alone results in excess copying and adjustment, and consequently, performance suffers due to an increase in old generation GC time by around 6% (or 0.07 sec, 0.16 sec and 0.70 sec for 2, 5 and 10 instances respectively).

For other benchmarks, we did not encounter a significant change in execution time (however, in all cases, hybrid GC never performs worse than using mark-sweep or copying alone).

To summarize, hybrid GC achieves a lower footprint in many cases for benchmarks that show significant old generation GC activity, while maintaining performance that is on par or better than using mark-sweep or copying alone.

4.4 Related Work

The techniques that we present herein build upon a body of related work in garbage collection and multi-tasking MRE research. We first discuss prior work that is related to our task-aware scavenging mechanism, followed by related work for hybrid collection.

4.4.1 Application-Aware Memory Management

Prior work includes per-task young generations and implements reclamation of young generations. It describes temporary dynamic extension of the young generation space. It does not, however, provide reclamation of per-task old generation areas without triggering a full GC. This is especially important for non-trivial tasks that utilize the old generation. More importantly, we provide the ability to collect per-task young generations without pausing all tasks, which leads to better scalability. In addition, the prior work requires scanning of dirty cards belonging to all tasks during scavenging. This makes scavenge dependent on the number of tasks, which inhibits

scalability. Our old generation regions provide precise tracking of regions of the old generation on a per-task basis, and the number of concurrent tasks. This allows us to only scan cards belonging to the GC trigger task.

Detlefs et al's garbage-first GC [34] splits the heap into regions, which can be independently collected to satisfy a soft pause-time limit. The authors employ bidirectional remembered sets between regions to allow any set of regions to be collected independently of the others. They use GCLABs, which are thread private allocation buffers that they use during GC time. Since their collection policy is concurrent, threads compete to perform an object copy. Since we assign regions on a per-task basis, in the common case, there is only one per-task thread performing promotion, without the need for synchronization.

Prior work on thread-specific heaps [37, 86] focuses on improving performance for an application by enabling garbage collection on a per-thread basis, to minimize synchronization between application threads. Although, this helps achieve performance isolation, our work is different in that there is no sharing of objects between tasks in MVM. Consequently, we can achieve better isolation since we do not need to track references between young generations. Thread-specific heap techniques can be combined with our scheme to provide further performance isolation. However, performance isolation constitutes only a part of our work. A significant goal is also to

accurately identify heap usage, and readily reclaim heap space upon task termination, without requiring collection, minor or major.

Other researchers have presented complementary schemes for reducing old to young generation scanning time during minor collection, which may be combined with our per-task card table scanning mechanism. Azagury et al present a scheme for combining card marking with remembered sets [6, 49] in the train collector [51]. They maintain a per-card remembered set that is updated during card scanning, so that the card does not have to be scanned repeatedly unless it is modified. Another complementary approach for reducing scanning time, is to use a 2-level card table, with coarse and fine grain cards [33]. This is especially lucrative for large heaps, since regions of the heap that do not include old to young generation pointers can be logged as a few coarser level cards and quickly skipped.

Dimpsey et al [35] discuss compaction avoidance that leverages two key concepts [60] – address ordered allocation, and wilderness preservation. These techniques minimize heap fragmentation, and consequently, the frequency of compaction. Since our allocation scheme is a bump pointer, we automatically ensure address ordered allocation. Our scheme does not require free lists to be maintained and rebuilt by a mark phase. In addition, the part of the old generation beyond the end of the last region acts as a wilderness region. We perform large object allocations from this region directly (instead of regions), thereby reducing fragmentation.

J-Kernel [96] employs protection domains (called tasks) at the language level at compile-time to isolate applications and to assign access rights and ownership to applications. J-Kernel consists of an extension to the Java Language Specification (JLS), a bytecode to bytecode compiler and a set of libraries that provide domain, rights, sharing and resource management mechanisms. Communication between tasks is via capabilities. Objects are shared by passing a pointer to a capability object through a "local RMI (remote method invocation" call. The capability object contains a direct reference to the shared object, and thus access to the shared object is through a level of indirection, and can therefore be revoked. This enables full reclamation of a task's objects when a task terminates. However, GC activity is not isolated to a task – a GC for any task will pause all tasks.

Luna [42] is an extension to J-Kernel that enables inter-task communication via special types. Arbitrary sharing between tasks is possible, yet safe termination is guaranteed. Special types are used for inter-task communication through object references, which undergo extra indirection at execution time. When a task terminates, remote references are invalidated and reclamation of the terminated task's objects is possible.

The KaffeOS [7] provides isolation and resource management for untrusted Java applications. The primary aim is to provide protection and isolate applications from each other, and to control resources on a per-application basis. The MVM concept of

an isolate is a much lighter-weight abstraction than a KaffeOS process, hence more efficient, albeit with fewer features. The garbage collector in the KaffeOS is non-generational and conservative. Consequently, there is no provision to handle correctness and efficiency on a per-process basis in the presence of modern GC techniques, such as pre-tenuring, or thread-local allocation areas. Our work employs a state-of-the-art generational collector with each task using its own separate young generation, but with the old generation shared across tasks to enable better scalability. In addition, we enable optimizations, such as fast reclamation of old generation areas upon task termination, regions and efficient card table scanning, in order to optimize throughput for modern multi-application environments.

Lastly, reclaiming heap resources on application termination does not require marking and tracing [72, 97] to identify per-task mature objects. By tracking promotion areas, we can readily reclaim all per-task dead mature objects upon task termination.

4.4.2 Scalable Hybrid Collection

To our knowledge, no prior work conclusively demonstrates that multi-tasking has the ability to outperform a single-tasking MRE in terms of execution time, as well as overall footprint for concurrent workloads (multiple applications executing

simultaneously). MVM is the most well known, state-of-the-art implementation of a multi-tasking MRE.

Prior work on MVM reports substantial improvement for startup, footprint and execution time compare to a corresponding single-tasking JVM [28, 29]. However, execution times were measured for serial execution of programs, and footprint of concurrently running programs were obtained when applications were quiescent, and do not reflect the footprint of programs when they are actually running concurrently and are exercising the memory management system.

Sun Microsystems' CLDC HotSpot Implementation, aimed at small hand-held devices, supports multi-tasking in a way that is similar to MVM, but uses a single heap shared by all tasks [90], with no provision for GC performance isolation.

Singularity [54] is a research operating system from Microsoft Research that uses type safety at the language level in order to attempt to achieve a dependable OS. Applications, extensions, services, device drivers and the kernel are written using safe languages. Only parts of the kernel employ unsafe code. The abstraction used to provide isolation between applications is the software isolation process (SIP), which consists of a runtime, libraries, application code and data. SIPs are isolated from each other at the language level by not allowing the same object to be accessed by multiple SIPs. Ownership of objects can be transferred using bidirectional communication channels. Application code is statically compiled when a SIP is composed.

Each SIP can execute its own, possible different garbage collector. System code is collected using concurrent mark-sweep. However, we were unable to find results that demonstrate performance of the memory management system in Singularity.

Our hybrid GC bears some similitude to incremental copying GCs that divide the heap into equally sized regions that can be evacuated independently of others. In our case, heap space partitioning is primarily motivated by the need to allocate private tenured space to isolated applications on demand. Like our hybrid GC, Garbage First [34] only evacuates regions that can be reclaimed with little copying. Information regarding the amount of live data in regions is provided by a concurrent marker (as opposed to a synchronous marking phase in our case). Bidirectional remembered sets between regions are maintained by mutators (with help from the concurrent marker) to allow any set of regions to be collected independently of the others. In the case of our hybrid GC, this property is achieved by gathering cross-regions connectivity information during marking. The Mature Object Space (MOS) collector of Hudson and Moss [53] is another region-based incremental copying GC. It uses unidirectional remembered sets, which requires regions to be evacuated in order. MOS cannot therefore pick an arbitrary region to evacuate based on cost-related criteria (e.g., amount of live data). Both Garbage First and MOS are evacuation-only GC.

Lang and Dupont [68] describe a hybrid mark-sweep and copy similar to ours. The heap is divided into equal size segments. During GC, a single segment is com-

pacted, while others are swept. Like our hybrid mark-sweep-evacuate GC, the collector is primarily mark-sweep. The cost of compaction is bounded since a single segment is collected. However, the segment compacted at each GC is chosen arbitrarily. By contrast, we use copying opportunistically, only to evacuate sparsely populated regions or highly fragmented one. We may thus evacuate several regions during a single GC, or none if the regions are densely populated with little fragmentation.

MC^2 [80] and its predecessor, Mark-Copy [79] describe an incremental copying GC that uses a marking phase to precisely annotate equal size regions of the old generation of the heap with the amount of live data in them, like our GC, and then build uni-directional remembered set to update pointers to evacuated objects. MC^2 builds precise remembered sets, whereas we build an imprecise connectivity matrix that only records regions that references other regions. MC^2 aims at achieving good throughput and low pause times for memory constrained devices.

Beltway [15] provides incremental and generational GC by partitioning the heap into *belts* and collecting a single belt during GC. Garbage cycles larger than a belt cannot be reclaimed by collecting a single belt. However, Beltway has a provision for performing full GC by providing a separate belt with a single region and collecting this when it occupies half the heap space. Our per-application GC is complete and reclaims all garbage for that application. We, therefore, do not require precise remembered sets between regions or need mechanisms to ensure completeness.

McGachey et al [73] present a scheme that uses a generational GC with a reduced copy reserve, with the ability to dynamically switch to a compacting GC if necessary.

Page unmapping as well as compaction has been used to reduce application memory footprint in prior work, such as the Compressor [64]. However, Compressor is a concurrent, parallel compacting GC that achieves low pause times. Our goal is different: to provide a relative simple, per-application GC that achieves good footprint and overall performance for desktop or small client applications, while allowing other applications to execute concurrently, without interference.

4.5 Summary

Multi-tasking has been proposed as a means to enable sharing of code and classes between applications in order to enable better startup performance, footprint and for faster overall execution compared to single-tasking, i.e., executing each application in a separate MRE process. While prior implementations of multi-tasking have demonstrated the above for serial execution of programs, we show that the prior state-of-the-art performs poorly for concurrent workloads. We attribute this to lack of performance isolation due to a heap layout and GC that is not amenable to scaling. In addition, prior work lacks precise resource accounting, the ability to manage mem-

ory subsystem setting (mainly heap size limits) on a per-application basis and GC-less reclamation of an application's resources on termination.

We present a series of generational memory management techniques to improve the efficiency and scalability of a multi-tasking virtual machine (MVM) for the Java programming language. Our techniques partition the young generation into per-task regions that are isolated from other tasks, track old generation heap consumption on a per-task basis, and facilitate concurrent mutation activity with garbage collection. These MVM extensions enable fine-grain control of application-specific heap parameterization and accounting, immediate reclamation of heap areas upon task termination, concurrent allocation in the young generation, promotion of objects during minor and major collection for only the task that triggers GC, and reduced scanning overhead during GC.

Further, we describe a hybrid GC for old generation collection that achieves scalable performance and low footprint. Our hybrid GC combines mark-sweep with copying collection in the same GC cycle along with fast adjustment for copied objects, to achieve good performance and a low footprint while avoiding the overhead of full copying GC. The hybrid GC uses marking to gather information (liveness, connectivity, occupancy, and estimated fragmentation) necessary to determine regions of the old generation to evacuate (if any) and to sweep and to identify which regions must be scanned for pointer adjustment.

We have integrated MTM with MVM, a multi-tasking implementation of the JVM, and a compare it to a widely used, production-quality, single-tasking MRE for concurrent application workloads. Our results show that MTM enables significant improvements in overall execution time, throughput as well as footprint for concurrent workloads, compared to prior state-of-the-art single- as well as multi-tasking MREs.

These results indicate that multi-tasking is a viable approach for executing concurrent applications and strengthens the case for multi-tasking MREs.

The text of chapter 4 is in part a reprint of the material as it appears in the proceedings of the International Symposium on Memory Management (ISMM 2006) and in the proceedings of the European Conference on Object Oriented Programming (ECOOP 2008). The dissertation author was the primary researcher and author and the co-authors listed directed and supervised the research that forms the basis for this chapter.

Chapter 5

Conclusion

The pervasive nature of heterogeneous networked computing platforms has made portability, security, and programmer productivity key concerns today. Today's applications are programmed in type-safe, object-oriented languages that provide language level features to enable productivity. Programs written in these high-level languages are translated into an intermediate architecture-neutral format and executed in execution environments, known as Managed Runtime Environments (MREs), that virtualize the underlying hardware architecture and resources for programs.

Modern MREs provide a number of runtime services that enable better productivity, security, and portability. Automatic memory management or garbage collection (GC) is one such service. Today's Internet programs do not use explicit memory allocation and de-allocation. Programmers rely on garbage collection to reclaim and reuse memory freed by programs at execution time. Since MREs perform garbage collection while the application is executing, they potentially imposes significant per-

formance overhead. Much prior work has attempted to mitigate the overhead and execution time impact of GC. However, users today execute a wide variety of different applications, ranging from cell phone programs to larger desk-side programs, at a time. Little attention by researchers has been directed at MREs that support different multi-program execution models.

In this dissertation, we examine garbage collection for MREs that execute multiple applications. We consider single-tasking persistent MREs that execute a single application in a single MRE instance (operating system process), as well as multi-tasking, persistent MREs that execute multiple applications concurrently in a single MRE instance in an effort to share application code and data structures.

With this thesis work, we find that due to the diversity of application characteristics and memory requirements, a single general-purpose GC algorithm does not enable the best performance for all applications and heap sizes. We propose that MREs be able to dynamically select GC algorithms at execution time. Moreover, we design, implement, and evaluate a GC switching framework that allows the GC algorithm to be changed at execution time.

We demonstrate two uses of our dynamic GC swapping framework – annotation-guided GC switching that selects the best performing GC for an application based on user-supplied annotations that are determined based on a priori profiling and validated across multiple inputs; and adaptive GC selection that attempts to achieve good per-

formance using a heuristic that is based on the heap size for an application and heap residency. We find that to achieve high performance, application code must be specialized based on the choice of the GC. Further, if the GC algorithm is changed at execution time, this code must be de-specialized by the MRE. Existing de-optimization approaches are inadequate since they do not provide a general-purpose mechanism to de-optimize code that is being executed without inserting special instructions in the application code as well as introducing register pressure for platforms with a limited number of registers. We, therefore, introduce a general-purpose de-optimization mechanism that tracks compiler optimizations and that maintains state information for de-optimization, that the MRE performs out-of-line with generated application code.

We demonstrate that significant improvement in execution time is possible by using a GC algorithm that is best suited to a particular application, the code of which is specialized for the GC algorithms. We refer to such an approach as *application-specific GC*.

We then investigate GC performance in multi-tasking MREs that execute multiple concurrent applications in the same MRE instance. To enable sharing of application code and data structures, multi-tasking MREs execute applications in the same address space. We find that state-of-the-art multi-tasking MREs suffer from lack of performance isolation between applications and lack precise resource account-

ing. Further, these MREs must perform GC to reclaim memory that was used by terminated applications. When executing concurrent applications, scalability of the memory management subsystem is a key concern.

To address these limitations, we present the *Multi-Tasking Memory Manager* (MTM) that enables performance isolation in a shared address space by providing each application with the view that it alone is executing in the MRE instance. We achieve this through on-demand allocation of heap memory regions, a synchronization mechanism that allows only a single application to be paused for GC, and leveraging existing generational GC mechanisms. Further, to manage heap footprint, yet enable high performance, we prototype a hybrid GC technique that combines two different GC algorithms – mark-sweep GC and copying GC, and makes a dynamic decision about which technique to apply to a particular old generation region. Copying GC mitigates fragmentation at the cost of requiring a copy reserve area and excessive copying of live data. Mark-sweep GC is fast and suitable for old generation collection since a majority of old generation objects are alive, however, it introduces fragmentation if there are holes created by dead objects, which cannot accommodate future allocations. The selection of the best garbage collection algorithm for performing collection of a given old generation region is, thus, *dynamic*.

Our techniques enables a state-of-the-art MRE, the Sun Microsystem Labs' Multi-tasking Virtual Machine (MVM), to provide scalable performance as well as a small

heap footprint for concurrent workloads. Our techniques also facilitate other improvements, such as GC-free reclamation of terminated applications' resources and selection of memory subsystem parameters on a per-application basis.

In summary, with this research and dissertation, we find that it is possible to enable dynamic GC selection on a per-application basis for single-tasking MREs. Further, we describe an effective implementation of scalable application-aware GC for multi-tasking MREs, as well as dynamic and adaptive selection of two different GC algorithms based on simple online heuristics (hybrid GC) to extract performance. We show that it is possible to enable high performance memory management for the next generation of multi-application environments and portable application technologies. Our findings and the contributions that we make with this dissertation significantly advance the state of the art of modern MRE systems. We next discuss potential directions for future research.

5.1 Future Work

In our work on application-specific GC for single-tasking MREs, we have implemented two simple heuristics to guide adaptive GC switching. Although these heuristics perform well, as part of future work, we plan to investigate the use of additional, hardware-level information, such as memory hierarchy and cache statistics.

Most modern CPUs provide hardware performance monitors for performance profiling. We plan to investigate the use of hardware monitors and dynamic profiling to guide GC selection.

The GC algorithms we experimented with in the first part of the thesis are stop-the-world, i.e., all mutators must be paused during GC. Stop-the-world GC is most commonly used for small to medium sized devices. However, for large servers with many cores, it may be beneficial to use parallel (multiple threads performing GC) and/or concurrent GC (executing GC concurrently with the application that triggered GC). As part of future work, we plan to investigate dynamic switching for GC algorithms that are more suited to server systems.

Our work on scalable memory management for multi-tasking MREs shows that it is possible to extract high performance from a state-of-the-art MRE; our techniques enables multi-tasking to perform to its potential. Our old generation GC is a hybrid mark-sweep-compact collector that reclaims discontinuous regions on a per-application basis. We show that performing mark-compact collection over the entire old generation is prohibitively expensive. Instead, we choose hybrid GC, which combines mark-sweep GC for regions without much fragmentation and free space and copying GC for fragmented and free regions. This achieves a good balance between performance and footprint. However, copying GC requires a copy reserve area. If memory is constrained and there exists significant fragmentation, copying/evacuation

is unsuitable. In this case, we fall back to a full heap compacting collection. As part of future work, we plan to investigate compacting collection over discontinuous regions for memory constrained devices.

In a multi-tasking MRE, the GC acts as a service that is common to all applications, i.e., the GC is aware of all applications executing in the MRE. If an application is idle, yet using memory, it might be beneficial to perform a collection for that application and reuse memory freed by it for other applications, even if the application does not itself trigger a GC.

Today's server (and many desktop) platforms support 64-bit address spaces. The availability of a large address space raises interesting questions about the address space layout for multi-tasking MREs. We can potentially provide each application with a single contiguous address space, yet enable sharing through a carefully designed address space layout. A per-application contiguous virtual address space can be mapped to discontinuous physical memory regions, enabling the GC to reclaim a contiguous region of memory per application. MRE design and memory layout for future massively multi-core platforms with a large amount of physical memory as well as a large virtual address space is an interesting future work area.

Bibliography

[1] A. Aiken and D. Gay. Memory management with explicit regions. In *Conference on Programming Language Design and Implementation*, May 1998.

[2] 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. *IBM Systems Journal*, 39(1):211–221, 2000.

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

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[5] C. R. Attanasio, D. F. Bacon, A. Cocchi, and S. Smith. A comparative evaluation of parallel garbage collectors. In *Fourteenth Annual Workshop on Languages and Compilers for Parallel Computing*, Cumberland Falls, KT, Aug. 2001. Springer-Verlag.

[6] A. Azagury, E. K. Kolodner, E. Petrank, and Z. Yehudai. Combining Card Marking with Remembered Sets: How to Save Scanning Time. In *International Symposium on Memory Management (ISMM)*, Oct. 1998.

[7] G. Back and W. C. Hsieh. The KaffeOS Java Runtime System. *ACM Trans. on Programming Languages and Systems*, 27(4):583–630, July 2005.

[8] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In B. Magnusson, editor, *Proceedings of the Sixteenth European Conference on Object-Oriented Programming*, volume 2374

of *Lecture Notes in Computer Science*, pages 111–132, Málaga, Spain, June 2002. Springer-Verlag.

[9] K. Barabash, Y. Ossia, and E. Petrank. Mostly Concurrent Garbage Collection Revisited. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2003.

[10] BEA Systems Inc. BEA's Enterprise Platform. IDC white paper by M. Rosen sponsered by BEA, 2002. `http://www.bea.com/content/news_events/white_papers/BEA_Beyond_Application_Server_wp.pdf`.

[11] BEA Systems Inc. BEA Weblogic JRockit: Java For the Enterprise, Dec 2003. `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[12] S. Blackburn, P. Cheng, and K. McKinley. A Garbage Collection Design and Bakeoff in JMTk: An Efficient Extensible Java Memory Management Toolkit. Technical Report TR-CS-03-02, Department of Computer Science, FEIT, ANU, Feb 2003. `http://eprints.anu.edu.au/archive/00001986/`.

[13] S. Blackburn and K. McKinley. In or Out? Putting Write Barriers in Their Place. In *International Symposium on Memory Management (ISMM)*, 2002.

[14] S. Blackburn, J. Moss, K. McKinley, and D. Stephanovic. Pretenuring for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Tampa, FL, Oct 2001.

[15] S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Conference on Programming Language Design and Implementation*, June 2002.

[16] S. M. Blackburn and K. S. McKinley. Ulterior referene counting: Fast garbage collection without a long wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, SIGPLAN Notices, Anaheim, CA, Nov. 2003. Association for Computing Machinery.

[17] D. Box. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley Professional, Nov. 2002.

[18] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce execution time of Java applications. In *ACM*

Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), Nov. 2001.

[19] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Shreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proceedings of the ACM Java Grande Conference*, pages 129–141, June 1999.

[20] B. Cahoon and K. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java Controller. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2001.

[21] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Objected-Oriented Programming Language*. PhD thesis, Stanford University, Mar. 1992.

[22] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–15, 1991.

[23] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[24] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[25] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, 1983.

[26] G. Czajkowski. Application Isolation in the Java™ Virtual Machine. In *OOPSLA*, pages 354–366, 2000.

[27] G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2001.

[28] G. Czajkowski and L. Daynès. Multitasking without Compromise: A Virtual Machine Evolution. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2001.

[29] G. Czajkowski and L. Daynès. A Multi-User Virtual Machine. In *USENIX 2003 Annual Technical Conference*, June 2003.

[30] G. Czajkowski, L. Daynès, and N. Nystrom. Code Sharing among Virtual Machines. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[31] The Dacapo Benchmark Suite, version beta050224. `http://www-ali.cs.umass.edu/DaCapo/gcbm.html`.

[32] L. Daynès and G. Czajkowski. Sharing the Runtime Representation of Classes Across Class Loaders. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2005.

[33] D. Detlefs, W. D. Clinger, and M. Jacob. Concurrent Remembered Set REfinement in Generational Garbage Collection. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'02)*, Aug. 2002.

[34] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-First Garbage Collection. In *International Symposium on Memory Management (ISMM)*, Oct. 2004.

[35] R. Dimpsey, R. Arora, and K. Kuiper. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1), 2000. `http://www.research.ibm.com/journal/sj/391/dimpsey.html`.

[36] A. Diwan, E. Moss, and R. Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *Conference on Programming Language Design and Implementation (PLDI)*, June 1992.

[37] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-Local Heaps for Java. In *International Symposium on Memory Management (ISMM)*, June 2002.

[38] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[39] S. Fink and F. Qian. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[40] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the second international symposium on Memory management*, pages 111–120. ACM Press, 2000.

[41] Groovy: An agile dynamic language for the Java Platform. `http://groovy.codehaus.org/`.

[42] C. Hawblitzel and T. von Eicken. Luna: A Flexible Java Protection System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[43] M. Hicks, L. Hornof, J. Moore, and S. Nettles. A study of large object spaces. In R. Jones, editor, *ISMM'98 Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *SIGPLAN Notices*, Vancouver, Oct. 1998. Association for Computing Machinery.

[44] M. Hicks, L. Hornof, J. Moore, and S. Nettles. A study of large object spaces. In *ISMM98*, Mar. 1999.

[45] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, Oct. 1993.

[46] U. Hölzle. Optimizing Dynamically Dispatched Calls with Run-Time Type Feedback. In *Conference on Programming Language Design and Implementation (PLDI)*, June 1994.

[47] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Conference on Programming Language Design and Implementation (PLDI)*, June 1992.

[48] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness With Performance. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1994.

[49] A. L. Hosking and R. L. Hudson. Remembered Sets Can Also Play Cards. In *OOPSLA '93 Workshop on Garbage Collection and Memory Management*, Sept. 1993.

[50] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1992.

[51] R. L. Hudson, R. Morrison, J. E. B. Moss, and D. S. Munro. Garbage Collecting The World: One Car At A Time. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 1997.

[52] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Y. Bekkers and J. Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637, pages 16–18, St Malo, France, Sept. 1992. Springer-Verlag.

[53] R. L. Hudson and J. E. B. Moss. Incremental Garbage Collection for Mature Objects. In *International Workshop on Memory Management (IWMM)*, 1992.

[54] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. D. Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, Oct. 2005.

[55] IBM Corporation. Persistent Reusable JVM. Project home page. `http://www.haifa.il.ibm.com/projects/systems/rs/persistent.html`.

[56] IBM WebSphere. The WebSphere Software Platform. Product home page, 2004. `http://www-3.ibm.com/software/info1/websphere/index.jsp`.

[57] Java Community Process. JSR-121: Application Isolation API Specification. `http://jcp.org/jsr/detail/121.jsp`.

[58] Java Grande Forum. `http://www.javagrande.org/`.

[59] IBM Jikes Research Virtual Machine (RVM). `http://www-124.ibm.com/developerworks/oss/jikesrvm`.

[60] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[61] R. E. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley and Sons, July 1996.

[62] JRuby: Java powered Ruby implementation. `http://jruby.codehaus.org/`.

[63] A. Kennedy and D. Syme. Combining generics, pre-compilation and sharing between software-based processes. In *Proceedings of the Second Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*, Venice, Italy, Jan. 2004.

[64] H. Kermany and E. Petrank. The Compressor: concurrent, incremental, and parallel compaction. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 354–363, 2006.

[65] C. Krintz. Coupling On-Line and Off-Line Profile Information to Improve Program Performance. In *International Symposium on Code Generation and Optimization (CGO)*, Mar. 2003.

[66] C. Krintz and B. Calder. Using Annotation to Reduce Dynamic Optimization Time. In *Conference on Programming Language Design and Implementation*, pages 156–167, June 2001.

[67] B. Lang and F. Dupont. Incremental incrementally compacting garbage collection. In *Proc. of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 253–263, St. Paul, Minnesota, 1987.

[68] B. Lang and F. Dupont. Incremental Incrementally Compacting Garbage Collection. In *Symposium on Interpreters and Interpretive Techniques*, 1987.

[69] D. Lea. A memory allocator, 1997. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[70] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, 1983.

[71] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[72] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Comm. of the ACM*, 3:184–195, 1960.

[73] P. McGachey and A. L. Hosking. Reducing Generational Copy Reserve Overhead with Fallback Compaction. In *International Symposium on Memory Management (ISMM)*, June 2006.

[74] F. L. Morris. A Time- and Space-Efficient Garbage Compaction Algorithm. *Communications of the ACM*, 21(8), 1978.

[75] NonStop Server for Java Software. Project home page. `http://nonstop.compaq.com/view.asp?IO=NSJAVAPD01`.

[76] OpenSolaris Project: OpenGrok. `http://opensolaris.org/os/project/opengrok/`.

[77] M. Paleczny, C. Vick, and C. Click. The Java HotSpot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium (JVM'01)*, Apr. 2001.

[78] T. Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Usenix Java Virtual Machine Research and Technology Symposium*, Monterey, California, Apr. 2001.

[79] N. Sachindran, J. Eliot, and B. Moss. Mark-copy: Fast Copying GC with less Space Overhead. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2003.

[80] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: High-performance Garbage Collection for Memory-constrained Environments. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Oct. 2004.

[81] P. Sansom. Combining single-space and two-space compacting garbage collectors. In R. Heldal, C. K. Holst, and P. Wadler, editors, *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 312–323, Portree, Scotland, 1992. Springer-Verlag.

[82] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the first international symposium on Memory management*, pages 68–78. ACM Press, 1998.

[83] S. Soman, L. Daynès, and C. Krintz. Task-Aware Garbage Collection in a Multi-Tasking Virtual Machine. In *International Symposium on Memory Management (ISMM)*, June 2006.

[84] Standard Performance Evaluation Corporation (SpecJVM98 and SpecJBB Benchmarks), 1998. `http://www.spec.org/`.

[85] SpecJVM'98 Benchmarks. `http://www.spec.org/osg/jvm98`.

[86] B. Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *International Symposium on Memory Management (ISMM)*, Oct. 2000.

[87] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *SIGPLAN Notices*, pages 370–381, Denver, CO, Oct. 1999. Association for Computing Machinery.

[88] T. Suganuma, T. Yasue, and T. Nakatani. A Region-Based Compilation Technique for a Java Just-In-Time Compiler. In *Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[89] Sun Microsystems. The Java HotSpot Performance Engine Architecure. Whitepaper. `http://java.sun.com/products/hotspot/whitepaper.html`.

[90] Sun Microsystems Inc. CLDC Hotspot^{TM} Implementation Architecture Guide. `http://java.sun.com/javame/reference/docs/cldc-hi-2.0-web/doc/architecture/html/MultiTasking.html`.

[91] Sun Microsystems Inc. The Java Hotspot Virtual Machine White Paper. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html`.

[92] Sun Microsystems Labs. Multitasking virtual machine. `http://mvm.dev.java.net`.

[93] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Notices*, 19(5):157–167, 1984.

[94] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Recalamation Algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, Apr 1992.

[95] D. Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Recalamation Algorithm. In *Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Apr. 1992.

[96] T. von Eicken, C.-C. Chang, G. Czajkowski, C. Hawblitzel, D. Hu, and D. Spoonhower. J-Kernel: A Capability-Based Operating System for Java. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, pages 369–393, 1999.

[97] P. R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, Sept. 1992.

[98] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceeding of the International Workshop on Memory Management*, Kinross Scotland (UK), 1995.

[99] P. R. Wilson and T. G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Notices*, 24(5):87–92, 1989.

[100] L. Zhang and C. Krintz. Profile-driven Code Unloading for Resource-Constrained JVMs. In *ACM International Conference on the Principles and Practice of Programming in Java*, June 2004.

[101] B. Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 87–98. ACM Press, 1990.