UNIVERSITY OF CALIFORNIA,
IRVINE

Safe and Efficient Hybrid Memory Management for Java

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Codruț Stancu

Dissertation Committee:
Professor Michael Franz, Chair
Professor Alex Nicolau
Professor Guoqing Xu

2015

DEDICATION

To my loving family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Prof. Michael Franz for his mentorship and support. Exploring the interesting world of compiler construction under his supervision has been one of the most fulfilling period of my life, and for that I am forever grateful.

I would like to thank Prof. Alex Nicolau and Prof. Guoqing Xu for accepting to serve on my committee. The thesis was greatly improved by their valuable feedback.

I am deeply grateful to Dr. Christian Wimmer with whom I worked closely on much of this dissertation. His continuous guidance, motivation, and insightful comments made this thesis possible. Further, I am very grateful to Dr. Stefan Brunthaler and Dr. Per Larsen for always analyzing my work with a critical eye and providing honest feedback.

I would like to thank Dr. Michael Bebenita for his mentorship during my internship at Mozilla. I am also very grateful to Dr. Peter Kessler. His vast knowledge of garbage collection techniques and discipline in writing clean code made my life much easier.

Lastly, I would like to thank my great colleagues that made graduate studies a wonderful experience. My special thanks go to, in the order of meeting them: Andrei Homescu, Christoph Kerschbaumer, Wei Zhang, Gülfem Savrun Yeniçeri, Chen Li, Stephen Crane, Eric Hennigan, Mark Murphy, Mohaned Qunaibit, Julian Lettner, Brian Belleville.

CURRICULUM VITAE

Codruț Stancu

EDUCATION

Doctor of Philosophy in Computer Science — **2015**
University of California, Ivine — *Irvine, California*

Master of Science in Computer Science — **2012**
University of California, Ivine — *Irvine, California*

Bachelor of Science in Software Engineering — **2009**
University of Craiova — *Craiova, Romania*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2010–2015**
University of California, Irvine — *Irvine, California*

PROFESSIONAL EXPERIENCE

Oracle Labs — **since June 2013**
Research Assistant — *Redwood City, California*

Mozilla Labs — **Summer 2012**
Research Engineer Intern — *Mountain View, California*

Caphyon Ltd. — **2009-2010**
Software Engineer — *Craiova, Romania*

IT Six Global Services — **2007-2009**
Software Engineer — *Craiova, Romania*

TEACHING EXPERIENCE

Teaching Assistant - Compilers and Interpreters — **Spring 2012**
University of California, Irvine — *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

- Codruț Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, Michael Franz. **Safe and Efficient Hybrid Memory Management for Java**, In *Proceedings of the 2015 International Symposium on Memory Management, (ISMM '15)*, 2015.
- Codruț Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, Michael Franz. **Comparing Points-to Static Analysis with Runtime Recorded Profiling Data**, In *Proceedings of the 11th International Conference on Principles and Practices of Programming on the Java Platform, (PPPJ '14)*, 2014.
- Codruț Stancu, Luis Bathen, Nikil Dutt, Alex Nicolau. **AVid: Annotation Driven Video Decoding for Hybrid Memories**, In *Proceedings of the IEEE 10th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, 2012.

ABSTRACT OF THE DISSERTATION

Safe and Efficient Hybrid Memory Management for Java

By

Codruț Stancu

Doctor of Philosophy in Computer Science

University of California, Irvine, 2015

Professor Michael Franz, Chair

Java uses automatic memory management, usually implemented as a garbage-collected heap. That lifts the burden of manually allocating and deallocating memory, but it can incur significant runtime overhead and increase the memory footprint of applications. We propose a hybrid memory management scheme that utilizes region-based memory management to deallocate objects automatically on region exits. Static program analysis detects allocation sites that are safe for region allocation, i.e., the static analysis proves that the objects allocated at such a site are not reachable after the region exit. A regular garbage-collected heap is used for objects that are not region allocatable.

The region allocation exploits the temporal locality of object allocation. Our analysis uses coarse-grain source code annotations to disambiguate objects with non-overlapping lifetimes, and maps them to different memory regions. Region-allocated memory does not require garbage collection as the regions are simply deallocated when they go out of scope. The region allocation technique is backed by a garbage collector that manages memory that is not region allocated.

We provide a detailed description of the analysis, provide experimental results showing that as much as 78% of the memory is region allocatable and discuss how our hybrid memory management system can be implemented efficiently with respect to both space and time.

Chapter 1

Introduction

Many memory intensive applications follow a regular execution pattern that can be divided into execution phases. For example, an application server responds to user requests; a database performs transactions; or a compiler applies optimization phases during compilation of a method. These applications allocate phase-local temporary memory that is used only for the duration of the phase. Such coarse-grain phases can be identified by the application developer with a minimum of effort.

Figure 1.1 shows a high level view of a web server architecture. The web server is executed on a JVM instance and the various web services are implemented as Java applications. Client applications make requests to the various web services. In the architecture of a web server there are several layers of transactional execution patterns. At a high level each web service responds to a certain request kind. At a low level each web service is organized in execution phases that access the data store to build the result. In the process of accessing data and building the result, web services allocate temporary objects whose lifetime is shorter than that of the execution phases. A VM knowledgeable of the application execution patterns can reclaim the temporary memory with no overhead.

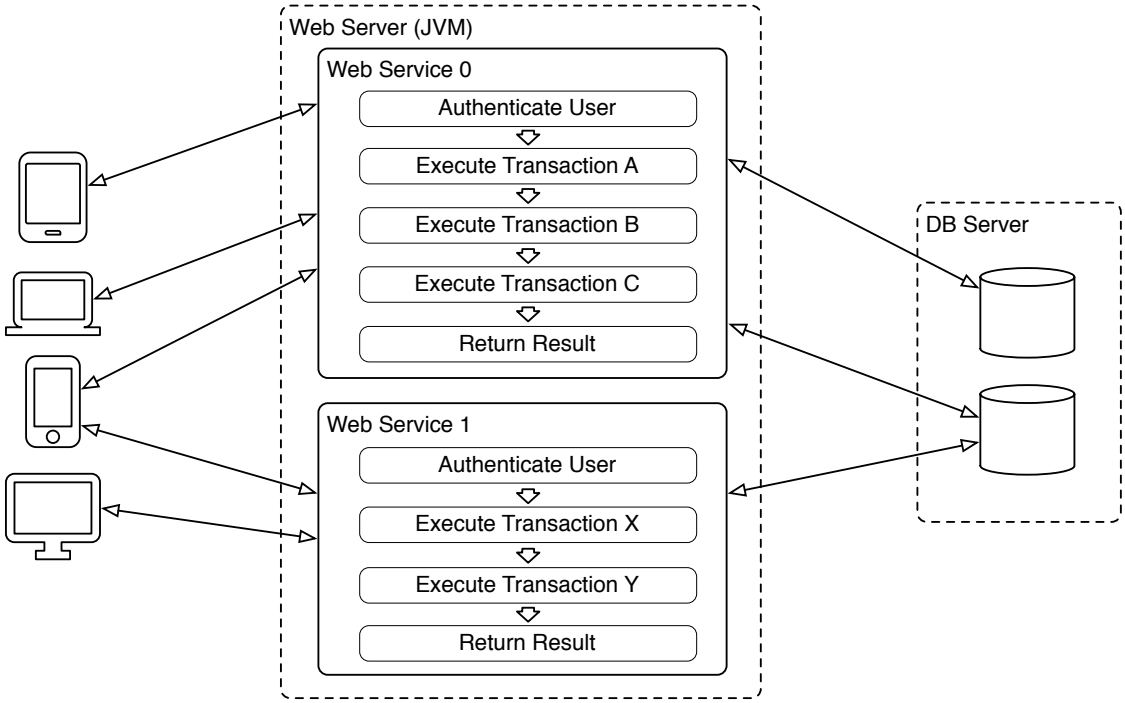

Figure 1.1: Web server architecture.

Given this class of applications the question is: how can the domain knowledge be used to optimize execution by reducing the memory management pressure? Our formulation of hybrid memory management tries to solve this problem by combining garbage collection (GC) with region-based memory management. The goal of hybrid memory management is to optimize the runtime resource utilization by reducing the time spent managing memory, i.e., reducing garbage collection time and reducing the number of garbage collections, and reduce memory footprint by deallocating objects earlier, before a GC.

Using source code annotations inserted by the programmer, each execution phase is mapped to a memory region. Objects that do not escape the execution phase where they are defined are allocated in the corresponding memory region. Objects escaping all memory regions are allocated into a garbage-collected heap. A region's lifetime starts when the application enters an execution phase and ends when the application leaves the execution phase, triggering deallocation by freeing memory. In general, the region-allocated memory can be deallocated without the overhead of garbage collection. Static analysis determines which allocation

sites are amenable to region allocation and enforces memory safety. Deallocating a region-allocated object cannot lead to dangling pointers, i.e., it is not possible that a region-allocated object is reachable after its region has exited. A region's size is not bounded and can increase dynamically to accommodate the object allocation. The region allocation scheme is complemented by a garbage collector which is triggered if the size of region-allocated memory grows beyond a configurable size. This enforces an upper bound on the total region-allocated memory and eliminates the possibility of excessive region sizes. In the worst case, if the programmer does a poor job annotating regions, our hybrid memory management behaves like a regular garbage-collected heap.

1.1 Thesis Contributions

We present a static analysis that maps allocation sites to memory regions, and evaluate the analysis on the industry standard SPECjbb2005 [50] benchmark. SPECjbb2005 is a memory intensive benchmark with a transactional execution pattern which makes it a good candidate for our intended scenario and gives an understanding of the potential of this technique. For the SPECjbb2005 benchmark, 78% of the total memory can be region allocated with a peak region memory size of less than 1 MByte. Our implementation is based on an ahead-of-time compilation system for Java that uses static analysis to also find all reachable methods. However, our findings are generally applicable for Java and other managed languages, since the static analysis for region-based memory management can be performed at runtime by a traditional Java VM, e.g., while the application is warming up, so that the just-in-time compiler can use the static analysis results.

In summary, this thesis contributes the following:

- We present a static program analysis that enables hybrid memory management based on source code annotations
- We show that our approach preserves memory safety and cannot lead to dangling pointers.
- We present the modifications of the allocator and garbage collector. The static analysis results can be reduced to one compile-time constant region offset per allocation site, minimizing the impact on allocation performance.
- We present empirical results for the SPECjbb2005 benchmark. The hybrid memory management reaches similar performance with a significantly smaller young generation when compared to a pure garbage-collected scheme.

1.2 Structure of the Thesis

Chapter 2, Background, sets the context for our hybrid, region-based and garbage-collected, memory management technique. It first gives an overview of points-to analyses formulations and introduces our memory region aware points-to analysis. Then it covers previous region-based memory management work and discuses differences to our approach. We also introduce the host system of our implementation, i.e., Substrate VM.

Chapter 3, System Structure, presents a high level view of our system, from region annotations to region analysis and runtime region management.

Chapter 4, Points-to Analysis in Substrate VM, discusses the details of the static analysis built in Substrate VM. This analysis is the basis of the later introduced region analysis.

Chapter 5, Static Analysis vs. Runtime Profiling, proposes a new methodology for static analysis evaluation, i.e., comparing its results with that of runtime profiling. Evaluating

the accuracy of the points-to analysis is an intermediary step in our study of region based memory management for Java.

Chapter 6, Points-to Analysis Evaluation, measures the accuracy of the Substrate VM points-to analysis results by comparing them with runtime profiling information extracted from HotSpot VM.

Portions of Chapter 5 and Chapter 6 were previously published in C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz, Comparing Points-to Static Analysis with Runtime Recorded Profiling Data, in Proceedings of the International Conference on the Principles and Practice of Programming in Java, 2014. [51].

Chapter 7, Memory Region Aware Points-to Analysis, introduces the memory region aware points-to analysis and discusses how it enhances efficient allocation and garbage collection while preserving safety.

Chapter 8, Hybrid Memory Management in Substrate VM, discuses the changes that were required to extend Substrate VM with region based memory management.

Chapter 9, Hybrid Memory Allocation Evaluation, evaluates the impact of hybrid memory management on execution time and memory used.

Portions of Chapter 7 and Chapter 9 were previously published in C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz, Safe and Efficient Hybrid Memory Management for Java, in Proceedings of the ACM International Symposium on Memory Management, 2015. [52].

Chapter 2

Background

2.1 Points-to Analysis

Static analysis is used to infer the runtime behavior of the code without knowledge about the input data. It abstracts the execution of the program conservatively to cover all execution scenarios. It uses abstract interpretation of the analyzed program statements operating on an abstract representation of runtime data.

We focus on object-oriented languages whose features and idioms, e.g., late function binding and encapsulation, make precise static analysis results hard to obtain [34]. The focus of our work is points-to analysis, a fundamental static analysis used by optimizing compilers to precisely determine the set of objects that a statement can access or modify at runtime. The points-to analysis is used as a basis for various optimizations such as virtual call or type check resolution. An efficient virtual call resolution must accurately infer the actual types of the receiver objects and is essential for constructing an accurate call graph. Our region analysis formulation extends the points-to analysis in new ways. It extends the abstraction with

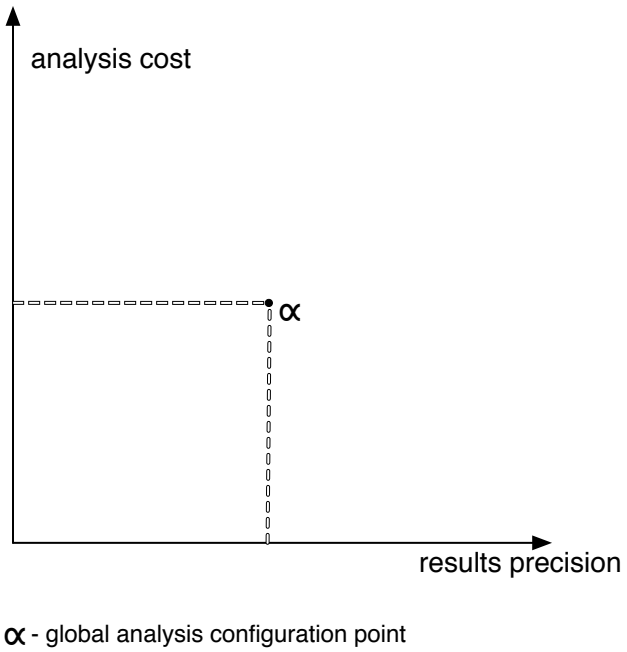

Figure 2.1: Points-to analysis design space.

domain knowledge to model more accurately an important aspect of runtime, i.e., memory management, with significant implications on performance.

The points-to analysis associates a points-to set to each reference variable and reference field. The elements of the points-to sets are abstractions of heap allocated objects. The analysis is modeled using flow transfer functions between program statements. In general, type state propagation happens at direct assignments, instance field reads and writes, and method invocations.

The two dimensions that define the design space of static analysis are *results precision* and *analysis cost* as shown in Figure 2.1. Choosing a design point in the static analysis space is equivalent to finding the trade-off between precision and cost best suited to the context in which the analysis results are used.

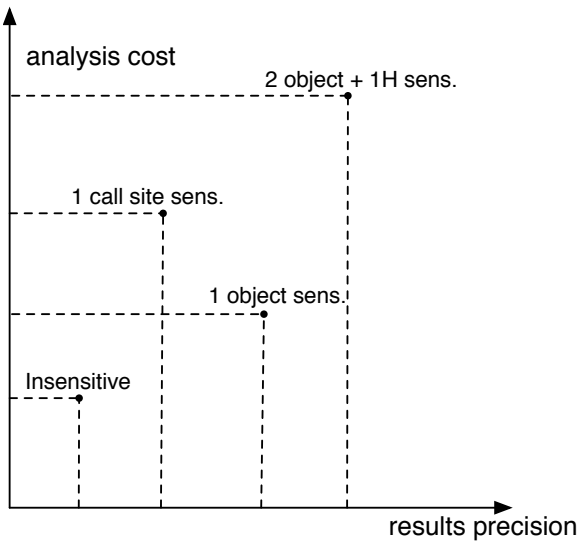

Figure 2.2: Points-to analyses classification. [34, 49]

Previous work has shown that an efficient points-to analysis for object-oriented languages like Java must be context sensitive, i.e., a method must be analyzed separately for each invocation context. Two of the most popular context choices used in literature are method invocation site, named *call-site sensitivity* [43, 46], and receiver object abstraction, named *object-sensitivity* [34].

From the introduction of object-sensitivity by Milanova et al. [34] it has been shown that it closely models the runtime behavior of object-oriented languages and yields a superior precision at a cost comparable to that of call-site sensitivity [9, 29, 30, 31, 35]. In addition, heap object abstractions can be modeled context sensitively by tagging object abstractions with the context of the allocator method; this is called heap sensitivity. Orthogonal to the choice of context is the context depth at which the analysis operates. That is, how many invocation levels are considered when grouping points-to sets of method variables or when abstracting heap allocated objects. A higher context depth yields more precise results but it reaches the limits of practicability quickly. Call-site sensitivity with a depth greater than 1 is typically considered impractical [34]. Object-sensitivity reaches the limit of practicality quickly too, at a depth of 2 with a heap sensitivity of 1 [49].

Figure 2.2 depicts the relative positioning of various points-to analyses formulations with respect to cost and results precision. The figure is compiled from previous results presented in context sensitive static analysis literature [34, 49] and is not to scale. It only serves the purpose of giving an understanding of how some context-sensitive formulations stand relative to the context-insensitive one. Our region analysis formulation extends the idea of context to encompass application domain specific information. The region borders can be interpreted as macro-context definitions which enable a classification of call paths that matches runtime execution patterns.

2.2 Region-Based Memory Management

Region-based memory management has been studied extensively [54]. Traditional region inference analysis translates the source language into a form with region-annotated expressions. The original program is instrumented with allocation and deallocation directives at compile time. The traditional region inference algorithm depends on the notion of region polymorphism, which allows region descriptors to be passed to functions at runtime. The passed region descriptors are used by value creating expressions to determine the allocation region.

Our program analysis takes a different approach in inferring the allocation regions and instrumenting the code for region allocation memory management. Our transformation does not require parameterizing methods with region descriptors. We instrument the allocation sites with statically determined allocation-site-to-region mappings. For instrumentation we use compile-time constants provided by the analysis such that the runtime can determine the concrete allocation region in a constant number of steps. Thus, our analysis formulation enables fast region allocation.

Unlike previous work [17, 18] that proposes explicit region constructs as language extensions for fine-grained region allocation, our approach uses annotations at a method granularity. The effort required to port application code that fits the described scenario is minimal, it is only necessary to identify coarse-grain application execution phases and insert annotations.

The relation between region allocation and garbage collection has been studied before by [20]. Their work is based on a fine grained region inference algorithm and uses region polymorphism. The specifics of our program analysis influence the relation between region-based and garbage-collected memory and opens the door to further optimizations. Unlike previous work, our technique deals with a relatively small number of larger regions. This can be seen as a disadvantage since a single region can potentially account for a large portion of the allocated memory, however it enables efficient region handling.

In particular, the programmer could annotate the entry point of a thread execution to get thread-local allocated memory. Multi-threaded applications are naturally organized in independent execution phases that share a limited amount a memory. Thus threads, or tasks executed by a thread pool, are a good candidate for coarse-grain region management. Similarly [53] proposes the use of escape analysis to enable allocation of thread-specific data in thread-specific heaps. This effects in independent garbage collection of data in thread-specific heaps, reducing garbage collection latency for active threads in a multi-threaded program and enabling concurrent garbage collection on multi-processor computers.

2.3 Substrate VM

The Substrate VM is low-footprint, fast start-up, embeddable virtual machine (VM) for and written in a subset of Java. It serves as an implementation basis for our hybrid memory management technique.

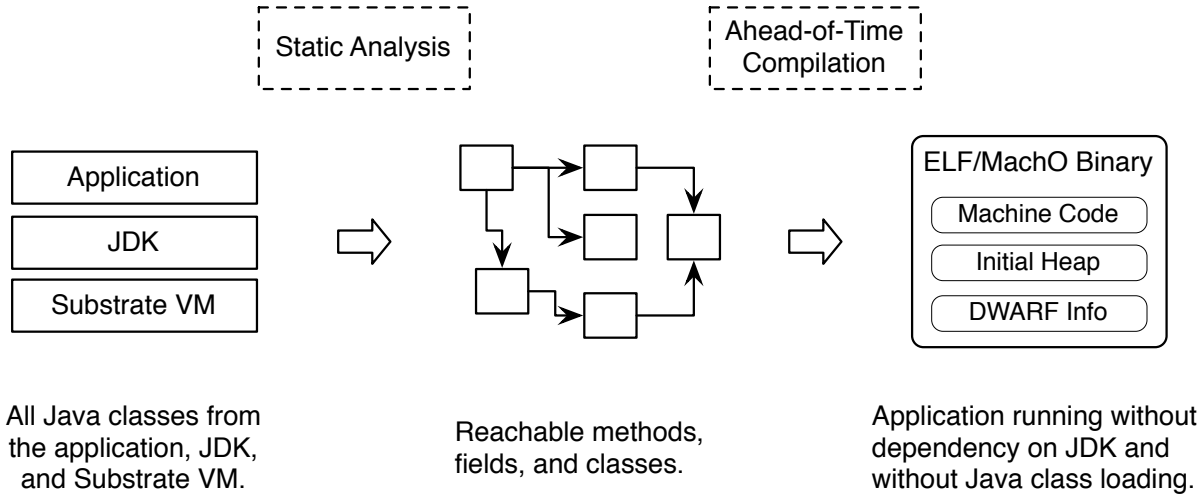

Figure 2.3: Substrate VM Execution Model.

Figure 2.3 depicts a high level view of the VM execution model. First the Substrate VM code is analyzed together with the Java Development Kit (JDK) libraries and application code. The static analysis discovers the runtime reachable world, i.e., methods, fields and classes, given a root method or a set of possible root methods. It reduces many virtual and interface calls to static calls. The compilation system is built around the Graal compiler [37]. It compiles the reachable world ahead-of-time to a single, self-contained binary. The binary contains all the code that it needs to run and it does not have any dependency on the JDK. Among the contained code is a generational garbage collector. The binary also contains the initial heap, i.e, the constant objects instantiated at compile time, and, optionally, DWARF debug information.

Java's dynamic features such as dynamic class loading and meta-programming capabilities such as reflection make ahead-of-time compilation difficult. Thus, in current iteration, the Substrate VM is built and supports applications written in a subset of Java. More specifically there is no dynamic class loading, there are no *ClassLoader* objects at run time and no access to *Class.forName()*. Consequently the Substrate VM binary does not include a bytecode parser. There is no lookup of methods, fields, and classes by name. The static analysis needs

to know the concrete target method at each call site. This limitations could be relaxed in the future and a limited amount of reflection could be supported.

The work in this thesis covers the static analysis step and it discusses how it was extended to support the region analysis. It also covers the modifications that were performed to the GC to efficiently support hybrid memory management.

2.3.1 Substrate VM Use Case

The characteristics of the Substrate VM, i.e., low footprint and low start-up time, enable usage of Java in not traditional scenarios. For example SVM can enhance the security of the web server discussed in Figure 1.1. Hosting the entire system of a single JVM can have serious security implications. If one of the clients is malicious and exploits a JVM/application bug that allows it to read memory of another client requesting the same web service it can conduct a data exfiltration attack.

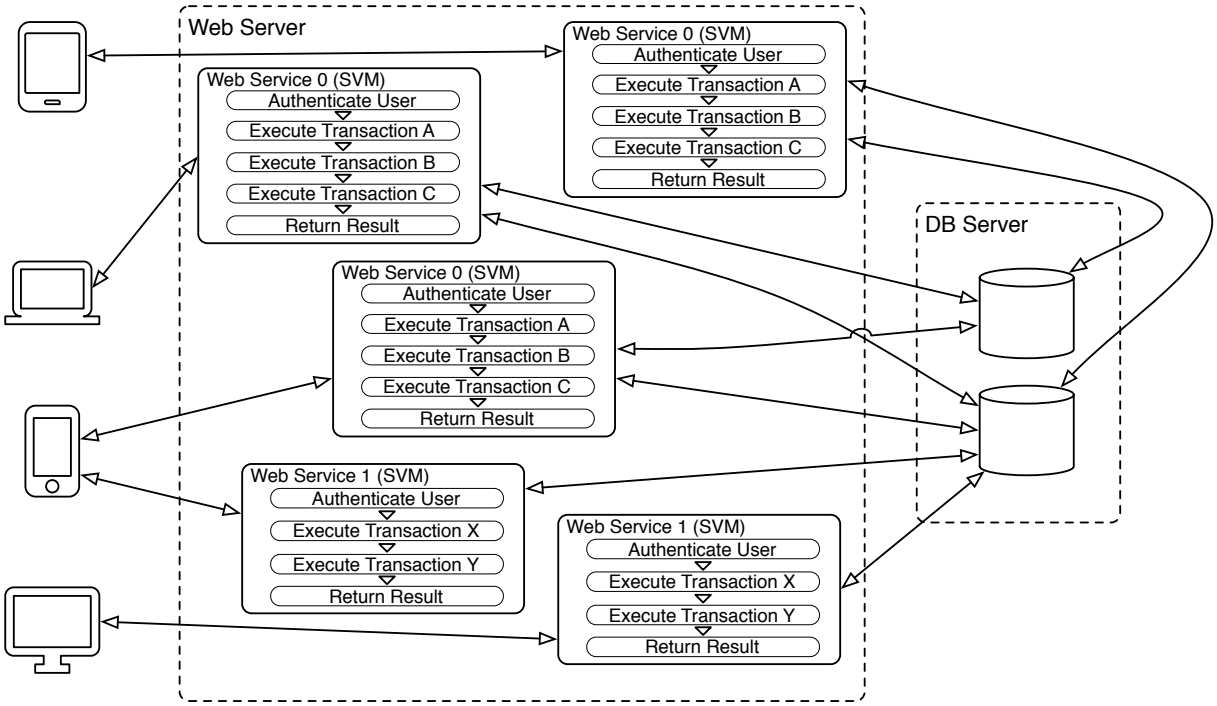

Figure 2.4: Web server on SVM.

Using SVM a defense strategy against such an attack can start a new VM instance for each service request as depicted in Figure 2.4. In this configuration the web server isolates each client request in a different memory space. The fast start-up ensures a low response delay and the low footprint guarantees a good resource utilization. Moreover, this strategy exploits the transactional execution pattern generated by the various independent requests. After the request is fulfilled the SVM instance is simply shutdown and the resources are reclaimed by the OS without any overhead.

Chapter 3

System Structure

In this chapter we present an overview of our system. We first describe the operation of the region analysis and its interaction with the points-to analysis. Then we present the main components of the region allocation and how it integrates with the garbage collector.

3.1 Region Identification

The region analysis relies on programmer-defined, coarse-grain annotations matching method borders as shown in Figure 3.1.

```
1 @Region(name = "foo-region")
2 public void foo() {
3     // ...
4 }
```

Figure 3.1: Annotation example.

The programmer annotates program points where the application enters execution phases that make significant use of temporary memory. The goal of the region analysis, given the

programmer-defined annotations, is to determine for each allocation site the runtime region in which it must be allocated.

3.2 Memory Region Aware Points-to Analysis

The region analysis is built as an extension to the context sensitive points-to analysis described in Chapter 4. Figure 3.2 displays a high level view of the interaction between the core points-to analysis and the region analysis.

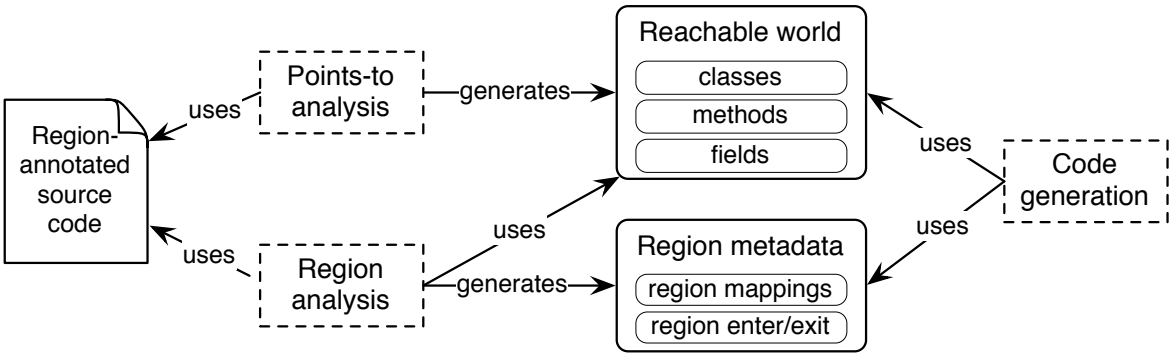

Figure 3.2: Region analysis overview.

The region annotations define application phases that start at the annotated method and contain all its callees. The region analysis processes the call graph discovered by the core analysis and tracks objects usage. It tries to prove that objects allocated inside a region don't escape it and thus can be region allocated. The inferred facts about allocation sites are summarized as region mapping tables which the code generator uses to perform the required allocation code modifications. The information regarding region borders is also embedded in the generated code and is used at runtime to automatically manage the region allocation and deallocation.

The full details of the memory region aware points-to analysis are presented in Chapter 7.

3.3 Hybrid Memory Management

The region allocation is build on top of a generational garbage collector. Traditionally a generational garbage collector divides the heap into young and old generations. New objects are allocated in the young generation until the allocation threshold is reached. When the allocation threshold is reached an incremental garbage collection is triggered which finds all live objects in the young generation and promotes them to the old generation. When the old generation allocation thershold is reached a full garbage collection is triggered.

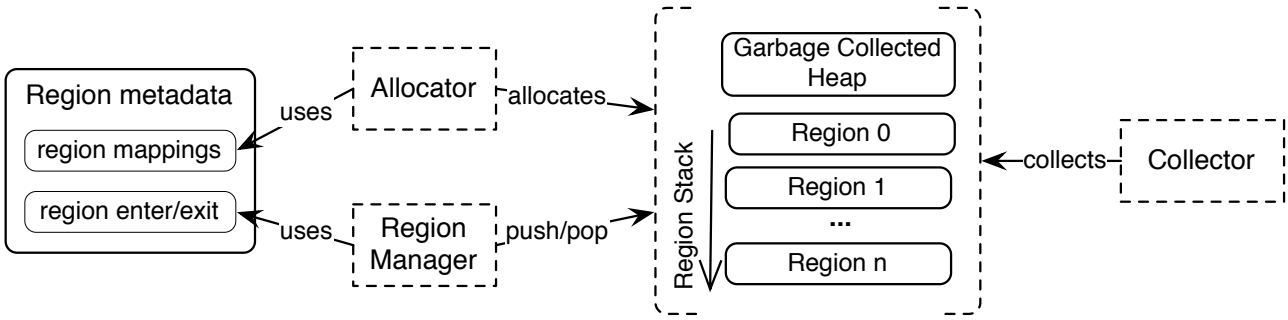

Figure 3.3: Hybrid memory management overview.

The region allocation extends the heap organization with a stack of regions, as shown in Figure 3.3. The region stack is conceptually part of the young generation. It is used for fast path allocation and it is scavenged together with the young generation. A region corresponds to an application phase and is used to allocate non-escaping objects in that phase.

To achieve efficient allocation the hybrid memory management scheme uses the region mappings inferred by the region analysis. When an object is allocated, the analysis results are queried to find the allocation region. The region manager is responsible for pushing and popping regions to and from the region stack. When a region-annotated method is called, a new region is pushed onto the stack. When a region-annotated method returns, its corresponding region is popped off the stack and its memory is freed without walking the objects.

The full details of the hybrid memory management are presented in Chapter 8.

Chapter 4

Points-to Analysis in Substrate VM

We implemented our entire system, including the points-to analysis, in Java. The core of our compilation system is the Graal compiler [37]. The points-to analysis phase is integrated in the compilation pipeline and it operates on the Graal internal representation [15]. The output of the analysis is summarized in data structures similar to Graal profiling summaries for later use in subsequent compilation stages.

4.1 Ahead-of-Time Compilation for Java

At a high level, the goal of the Substrate VM project is to ahead-of-time compile Java code to machine code. This decision is mainly motivated by the limited resources of the target machine. Our solution is to identify methods that are reachable using points-to analysis, covering application code, third party libraries, and the entire Java Development Kit (JDK).

Our system does not support reflection and dynamic class loading. These dynamic features cannot be supported for two reasons. First, the resulting executable binary does not contain code to load and link Java classes, therefore all the code must be available ahead of time.

Second, in presence of dynamic class loading and reflection the points-to analysis would need to make overly conservative assumptions. These assumptions greatly increase the size of the code and defeat our goals. There are various solutions that could be used to address these limitations and increase the spectrum of the supported language features. We cover those in the related work section.

4.2 Points-to Analysis

Our implementation follows closely the rules described by Smaragdakis et al. [49], a state of the art specification to points-to static analysis for object-oriented languages. Our points-to analysis implementation is context sensitive and flow insensitive. It implements a context-sensitive heap abstraction, i.e., it distinguishes between allocation sites of an object in different contexts. It is field-sensitive, i.e., distinguishes between different fields of an object and distinguishes the fields among different objects. It is array sensitive in the sense that it distinguishes between the elements points-to sets of different array objects, however, it is array-element insensitive, it does not consider different points to sets for each index of an array object. It is subset-based, preserving the directionality of assignments unlike equivalent-based analysis. It discovers the reachable world on-the-fly using a fixed-point approach. We define the reachable world as the call graph plus the collection of fields that are read or written.

4.2.1 Execution Model

At a macro level the analysis evolves in an iterative manner. The first iteration starts analyzing the entry method, e.g., the `main` method of the application. When the first iteration reaches a fixed-point, the analysis pauses and the newly discovered classes and

their constant pools are added to the analysis space. Then the analysis resumes and updates the discovered universe according to the newly discovered facts. This execution pattern evolves until the analysis reaches a global fixed-point.

At a micro level the analysis operates on a data flow graph built on top of the Graal intermediate representation (IR) [15]. Each IR node has an associated abstract object state. The abstract object state represents the collection of objects that flow through the instruction that the node models. For example, the object state of an allocation node contains the abstraction of the objects that can be generated by that node. The object abstraction contains at least information about the allocations site location (method and bci) and the declared type. In the context sensitive formulation of the analysis it can also contain allocator method context information. Two data dependent nodes are connected through an object flow edge. When the state of a data flow node changes the updated state is also propagated to its uses. For example, the object state of a virtual invoke contains the abstractions of the receiver objects flowing into that invoke through the various call paths.

Propagation of state between nodes is implemented as set operations. A node merges an incoming state with its current state using a set union operation. Nodes that model *checkcast* and *instanceof* operations are followed by a filter node which implements a set intersection operation. The filter node eliminates the abstract objects of types that wouldn't pass through the check at runtime, thus reducing the size of the type sets for dependent nodes. At runtime, incompatible types can still flow in the type check. The abstract objects are organized in unique type sets, thus redundant object state propagation is easily avoided.

The analysis implementation is built with scalability in mind. The abstract object state propagation is asynchronous and is implemented using synchronization free Java tasks. Each task takes care of propagating the state of the current data flow node to its uses. We use a thread pool executor to span the scheduled data flow updates in parallel on all available hardware threads as depicted in Figure 4.1.

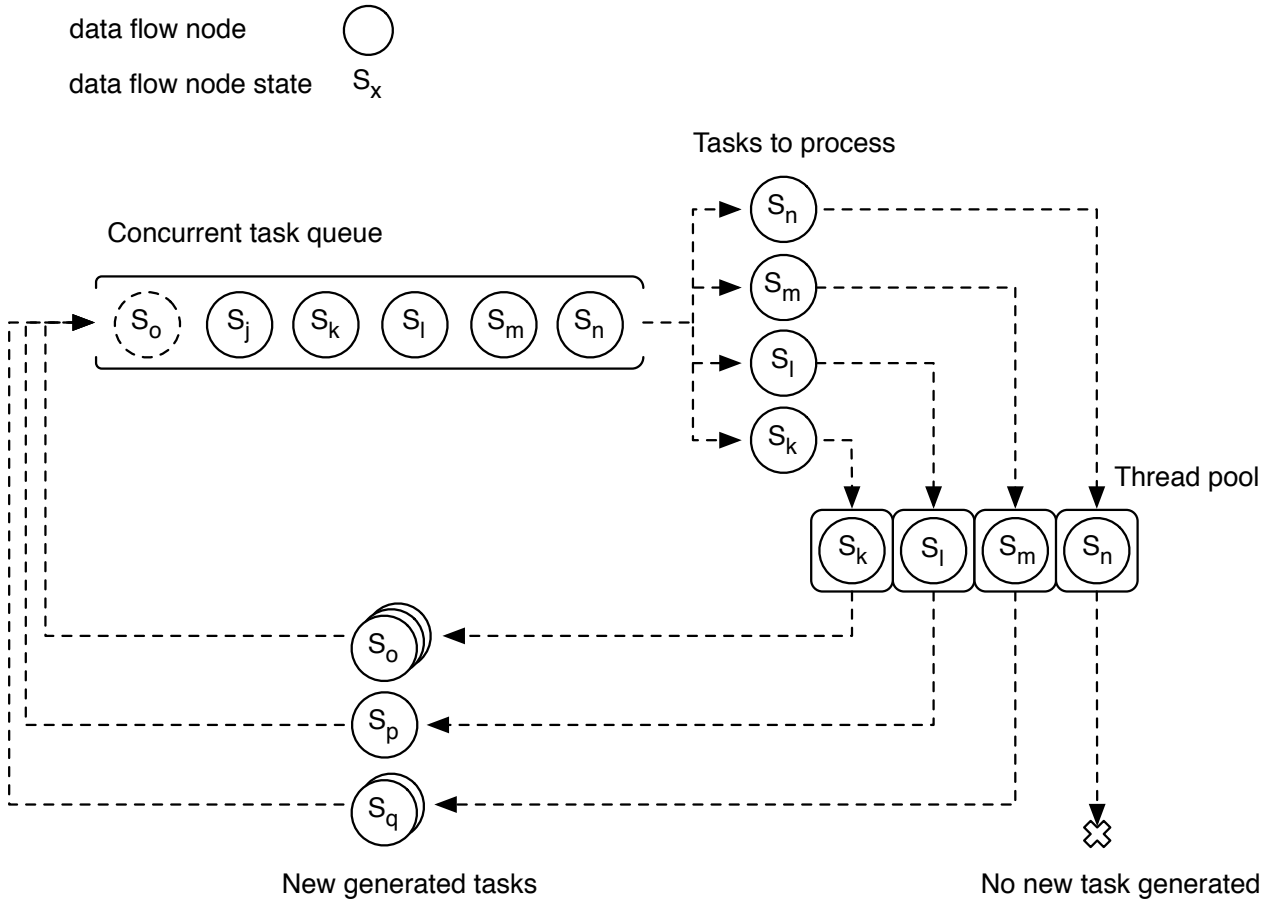

Figure 4.1: Analysis execution model.

When the state of a data flow node changes a new task is inserted at the tail of the concurrent task queue. The thread pool executor extracts scheduled tasks from the head of the queue and executes them. If the state of one of the uses changes then that state needs to be further propagated down the data flow graphs, thus a new task is inserted into the queue. If no data flow state change occurs then no new task is generated. The execution reaches a fixed point when the state of all data flow nodes is stable.

4.2.2 Context Sensitivity

We implement context sensitivity using function cloning, i.e., analyze each function separately for each different context. We only use the abstract value of the receiver object as

context, and not the value of all parameters. As stated by Milanova et al. [34] this is a good enough approximation for object-oriented languages. Creating a new clone for every distinct receiver object abstraction avoids the redundancy of creating a clone for each distinct call path.

Cloning a method is equivalent to duplicating its data flow graph and linking it to the call site. The actual parameters are linked to the formal parameters and the actual return is linked to the formal return. For efficient cloning we keep the original data flow graph in an uninitialized state, i.e., each data flow node has an empty object state. Thus the translation from Graal IR to analysis data flow nodes is only done once, when the method is first discovered. Every time the method is analyzed in a new context the original graph is duplicated and initialized. The initial object state of the graph comes from the input sources: allocation nodes, formal parameters, constants, static field reads.

Orthogonal to object sensitivity is heap sensitivity. To support heap sensitivity the object abstraction is extended with the context of its allocator method. The allocation nodes are cloned as a part of the allocator method cloning process. When the state of the new clone is initialized, the allocation node injects new abstract objects in the data flow graph corresponding to the new context.

4.3 Choice Of Input: Graal IR vs Raw Java Bytecode

Our points-to analysis implementation takes as input the internal representation generated by the Graal compiler. Using the Graal IR instead of the raw Java bytecode has several advantages.

First, Graal discovers early the trivially statically bindable virtual calls, i.e., calls to final methods or to methods declared in final classes. Additionally it can optimize type checks. For

example it removes type checks that always hold or always fail based on a simple inspection of receiver object declared type and the condition type. It can also fold two sequential checkcast instructions into one that checks for the more specific type if the first one has a less precise and compatible type. In the evaluation section we report the trivially statically bindable calls and the removed type checks separately and not as facts discovered by the analysis.

Second, Graal evaluates the constant objects when it parses the bytecodes and presents those as concrete Java objects to the analysis. Hence, the Graal IR gives a more concrete and easier to use representation of the analyzed code than the bytecode. The bytecode preprocessing step reduces the code complexity and enables the points-to analysis to save some iterations.

Chapter 5

Static Analysis vs. Runtime Profiling

In this chapter we shed new light on static analysis by studying it in comparison with runtime profiling. The benefits are two fold. First, static analysis and runtime profiling serve the same goal, that of inferring future application behavior. One uses facts derived conservatively from a static application abstraction, the other facts observed in recent application execution. However, the two techniques are in general treated separately and use different sources of information. We believe that the two techniques can leverage each other to achieve more accurate projections of future application executions. Second, most of the previous work on points-to analysis evaluates the accuracy of various analysis formulations using a context insensitive analysis as a base of comparison. By shifting the base of comparison to runtime profiling we want to understand what is the optimization potential of points-to analysis formulations when targeting performance. At the same time, static analysis inferred facts can optimize runtime profiling by inserting upper bounds on the types that can flow through the code.

5.1 Introduction

Java is a traditionally just-in-time (JIT) compiled language. Having access to concrete information about the application execution patterns, the JIT compiler improves performance through optimization of frequently executed code. Ahead-of-time (AOT) compilation is usually less efficient due to the language dynamic features. One of our work's motivations is to make AOT compilation for Java *efficient*. The first challenge in reaching this goal is to statically determine the reachable methods and inferring as much as possible about the application's runtime behavior. We rely on points-to static analysis to discover the application, Java standard libraries, and third party libraries reachable code and discard the rest. The precision is however limited. Prior work on Java static analysis [49, 27, 34] shows that the analysis can quickly become intractable for real world applications. In designing a static analysis one must find the trade-off between analysis cost, execution time and memory use, and results precision that is best suited to their context.

Static analysis uses various information to statically model application behavior over all possible executions. In particular, points-to analysis uses the object allocation site correlated with various context refinements such as call stack, object recency, heap connectivity information, and enclosing type [27, 32]. However, it traditionally ignores one important source: runtime information such as method execution frequency and concrete receiver types. When targeting performance this knowledge is an essential source of optimization.

A static analysis unaware of dynamic code behavior tries to infer facts with the highest precision by allocating equal resources over the entire code space. However, it is known that programs follow the 90/10 rule, that is, 90% of the execution time is spent in only 10% of the code. From a performance perspective the 90/10 rule suggests that inferring precise information for the 90% of *cold* code only has marginal utility. To maximize application performance an analysis should focus on increasing the precision of frequently executed

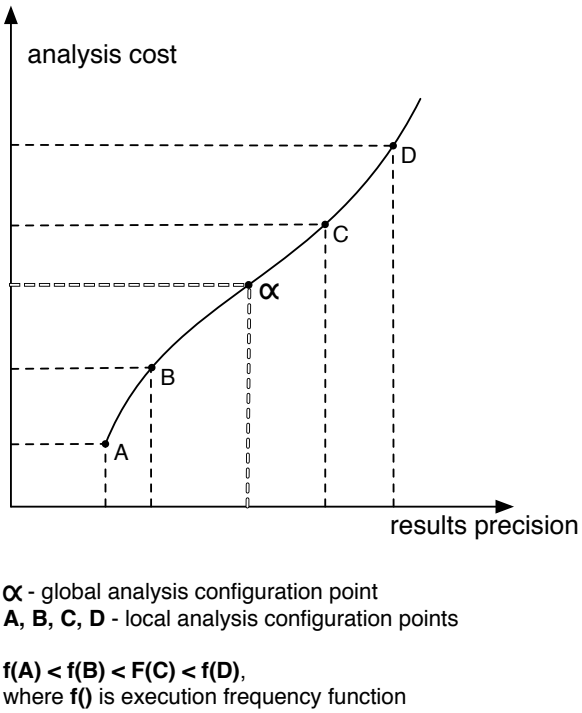

Figure 5.1: Demand-driven points-to analysis design space.

code. Taking into account execution frequencies the analysis can automatically find the best suited precision/cost trade-offs at a finer granularity. An adaptive analysis approach can pick different abstraction refinements and precision settings for different parts of the application.

Trading precision for cost using a fixed point in the design space for the entire application is rigid when targeting application performance. Ideally the static analysis would flexibly allocate more resources (i.e., use a more accurate object abstraction or an increased context depth) to obtain more accurate results for frequently executed code and analyze the rest of the application less precise (i.e., fall back to call-site sensitivity or disable heap sensitivity) to balance cost. To make an informed decision the static analysis requires knowledge about the runtime behavior of the code. It needs to know what the hot spots of the application are. Hence the design space for static analysis needs a third dimension, execution frequency. This knowledge can be used to automatically increase precision for frequently executed code at a cost penalty and to estimate the impact of results precision on runtime performance.

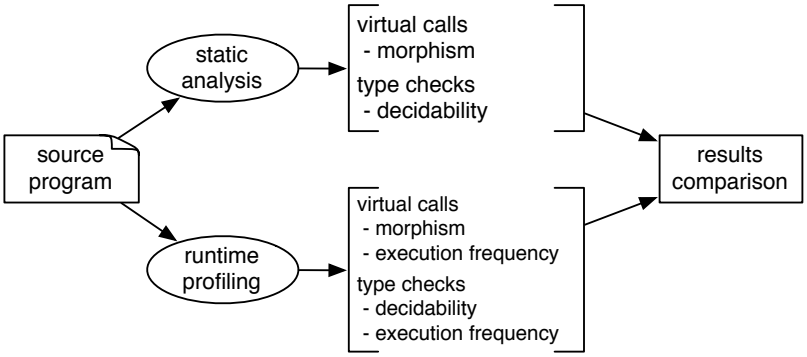

Figure 5.2: System overview.

As shown in Figure 5.1 the execution frequency function determines local configuration points that vary from the global configuration point adaptively, trading analysis cost for results precision and vice versa. Selectively varying the context abstraction and depth throughout the program requires the use of a demand-driven analysis.

In this study we identify the differences and similarities between static analysis and runtime profiling and discuss how they complement each other. Our system targets the Java programming language, however, the technique can be adopted by other languages as well. Figure 5.2 shows an overview of the proposed methodology. We begin by analyzing the code using our points-to analysis implementation for Java and extract analysis inferred facts. The analysis is purely static, does not use any dynamic information. We execute the same code in the Java HotSpot VM [38] and extract runtime profiling information collected by the interpreter. Next we compare the static analysis and runtime profiling information and interpret the results.

We extract virtual calls degree of morphism, i.e., number of resolved methods, and type check decidability, i.e., does the type check always hold or fail, from both static analysis and runtime profiling. Accurate information about virtual calls and type check receiver types is essential in optimizing object-oriented languages. Additionally we extract execution frequencies from runtime profiling. We use these metrics to inspect the analysis precision distribution over different execution frequency ranges and to project the impact of the static

analysis accuracy on runtime performance, focusing the attention on frequently executed code.

The contributions of our study are as follows:

- We discuss the importance of using runtime profiling information as an additional source of information to statically model application execution when motivated by performance.
- We present an empirical study that measures the precision of a points-to static analysis by comparing the results with runtime extracted profiling data.
- We find that the runtime profile is able to decide that 10% more frequently executed virtual calls are monomorphic and that 73% more frequently executed type checks are decidable when comparing to static analysis results.

5.2 A Motivating Example

Figure 5.4 shows an example of type check optimization based on static analysis inferred facts. The data flow graph is an abstract representation of the code in Figure 5.3. Method `foo` takes a parameter of type `Object` and returns an object of type `I`. Let's assume that `I` is an interface that is implemented by two classes `A` and `B`. The inputs in the data flow

```
1 static I foo(Object param) {
2   Object result = param;
3   if (...) {
4     result = new A();
5   }
6   return (I) result;
7 }
```

Figure 5.3: Code example

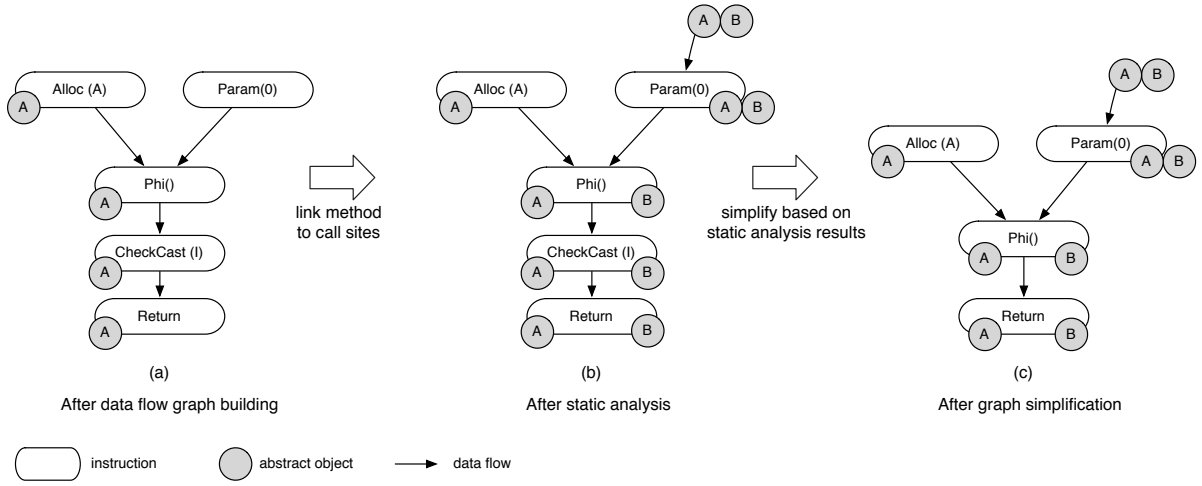

Figure 5.4: Static analysis data flow graph.

graph come from the object creation node (`Alloc(A)`) and from the formal parameter node (`Param(0)`). The control flow graph on which the data flow is built on is in static single assignment (SSA) form [14], which means that for every variable there is just a single point in the program where a value is assigned to it. The type sets of the two input nodes, coming from the two branches of a conditional statement, are merged in the `Phi()` node. In SSA form `Phi()` functions are placed at control-flow joins to indicate that the value of variables may come from either of the flow paths and it creates a new point of definition for every variable that is modified on either of the paths.

Figure 5.4(a) depicts the object states after data flow graph building. The state of the node which allocates `A` contains an abstract object of type `A`. The state of the allocation node is propagated to the `Phi()` node, it passes through the `CheckCast` statement since the two types are compatible and flows into the return type of `foo`.

When `foo` gets linked to a call site, Figure 5.4(b), the object state of the formal parameter node `Param(0)` gets updated to the object state of the actual parameter, abstract objects of type `A` and `B` in this case. This leads to a chain of object state updates. The `Phi()` node state is a union of its previous state and the newly incoming state. Since the `Phi()` node state is updated it propagates the new state further through the `CheckCast` to the `Return`

node. Assuming that no abstract objects of other types flow into the `Param(0)` the static analysis infers that the `CheckCast` node can be removed and that `foo` can return only types `A` and `B`, as shown in Figure 5.4(c).

A cheap but imprecise class hierarchy analysis based on the statically declared types would have failed to detect that the `CheckCast` can be removed and would have inferred that `foo` can return any `Object`. A more precise object-sensitive analysis could as well fail to optimize this code if objects incompatible with type `I` flow in `Param(0)` due to limitations in the maximum context depth. If `foo` is a frequently executed method the analysis should allocate best effort (e.g., use a more precise context abstraction and a higher context depth) to optimize it. Failing to disambiguate the types that flow into `Param(0)` would otherwise result in a penalty on runtime performance.

5.3 Runtime Profiling

Feedback-directed optimizations are commonly used together with dynamic compilation. The code is transformed at runtime using recently recorded execution profiles. The profiling information can range from method calls and backward branch execution counts to concrete types of objects.

In general the code is initially executed using an interpreter or baseline compiler that is instrumented for profiling. Once the profiling information reaches maturity, i.e., enough samples have been collected and the execution count of a call or backward branch has reached a certain threshold, the code is compiled to an optimized version. However, the profiling phase has a limited lifetime and the collected information reflects only the facts recorded up until the profile reaches maturity. Dynamic profiling can formulate optimistic assumptions about the code based on the recorded facts, but it cannot guarantee that those assumptions

hold for the entire duration of the execution. Thus the compiler must insert deoptimization guards that verify the correctness of the assumptions. In the event that the assumptions fail the guards trigger deoptimization and recompilation. Deoptimization is done by halting the execution of the compiled code containing the failing assumption and resuming execution in baseline mode where the exceptional case is executed and profiling restarted. Deoptimization requires a mechanism for transferring the execution state from optimized compiled code to baseline code.

Runtime feedback is especially important for object-oriented languages. Apart from traditional compiler optimizations, efficient compilation of object-oriented languages must use optimizations that target specific features and idioms that come with object orientation. Heavy use of polymorphic class hierarchies can impact performance unless special care is taken to optimize virtual calls.

Although there are special cases of virtual calls that can be statically dispatched, i.e., calls to methods that can be unambiguously resolved such as final methods or methods of final classes, virtual calls are in general dynamically dispatched based on the concrete types of receiver objects. Thus method binding is usually deferred to runtime when the concrete types of receiver objects can be recorded. The compiler constructs a polymorphic inline cache (PIC) that dispatches the call to the right callee based on the recently recorded receiver types [24]. The compiler can decide to apply aggressive optimizations based on the optimistic assumption that no other types flow into the receiver. It could for example decide to inline the target of a frequently executed virtual call. Inlining reduces function call overhead and provides opportunities for other compiler optimizations by increasing the context, i.e., size of continuous code. However, the compiler cannot guarantee that the assumption always holds so it has to insert guards that transfer the control to a runtime routine that patches the call site and potentially leads to deoptimization. Deoptimization occurring inside inlined

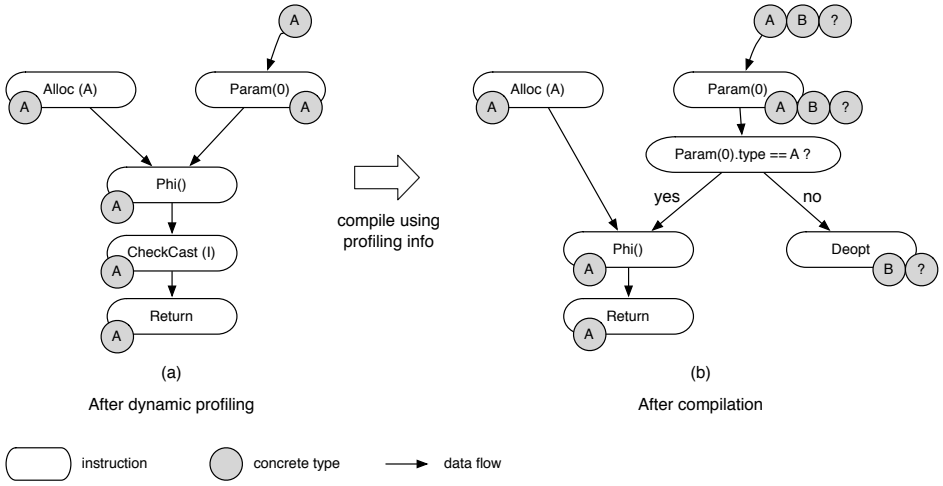

Figure 5.5: Runtime profiling data flow graph

methods is especially complicated since the compiler has to keep track of virtual call stack frames in order to reconstruct the interpreter native call stack.

Despite the fact that runtime profiling has only a statistical accuracy, yielding a view of the application runtime behavior that is limited by the span of the interpretative execution phase and the space allocated for profile metadata, it drives efficient compilation decisions [24]. However, to find virtual and interface calls that can be statically bound and type checks that can be statically decidable, compilers use a simple class hierarchy analysis (CHA) [39]. Enhancing the dynamic compilation with a more precise analysis would reduce the necessity of using guards and would enable more aggressive optimizations. Furthermore, current dynamic compilation techniques make method inlining decisions without distinguishing the profiling meta data for different invocation sites. Using a context sensitive dynamic profiling can lead to better inlining decisions.

5.4 Example Continued

Figure 5.5 reexamines the example of type check optimization, this time based on runtime profiling. The runtime profile cannot infer that the `CheckCast` node can be removed by simply inspecting the types of the values that flow in. Although for the duration of the profile the `CheckCast` may never fail, the compiler must still insert a type check guard.

However, the runtime profile has access to concrete runtime values. Assume that for the duration of the profiling the type of `Param(0)` is always `A`, as in Figure 5.5(a). This leads to an optimization that inserts a guard checking the exact type of `Param(0)` and removes the `CheckCast` node, as shown in Figure 5.5(b).

The exact type check guard is cheaper than the full dynamic type check that the `CheckCast` node would have performed. If the assumption fails, i.e., the objects that flow into `Param(0)` are of type `B` or a type that was not recorded during profiling, the guard triggers deoptimization. Inferring that `foo` can only return objects of type `A` can lead to further optimizations in `foo`'s caller too. For example, if `foo` is inlined into its caller method, the compiler can propagate the more precise type information within the caller.

Although the profiling information is not more accurate than the static analysis, it has access to concrete runtime objects and can drive more aggressive optimizations. However, with the additional knowledge offered by the static analysis the code could have been compiled to a more optimized version. The `CheckCast` could have been completely removed and no deoptimization guard would have been necessary.

Chapter 6

Points-to Analysis Evaluation

We implemented our whole-program points-to analysis in Java and extracted profiling metadata from the HotSpot VM. By comparing the two sets of results we want to answer the question: how much does increasing the analysis accuracy matter in the real world? Or, in other words, what would be the impact of increased points-to analysis accuracy on application performance?

6.1 Experimental Setup

We carefully configure the HotSpot VM execution to increase profiling information accuracy. We enabled interpreter profiling and disabled tiered compilation so that profiling happens only in the interpreter, without profile update in the client compiler (options `-XX:-TieredCompilation` and `-XX:+ProfileInterpreter`). We increased the compilation threshold to extend the interpreter execution phase (option `-XX:CompileThreshold=100000`). We also increased the values of the parameters that control the size of the collected profiles (options `-XX:TypeProfileWidth=10` and `-XX:MethodProfileWidth=10`). We run a HotSpot

VM build that includes the Graal extended profiling capabilities for more detailed profiling information.

We selected a variety of Java benchmarks to show how the two approaches compare in different scenarios. We analyzed:

- a number of DaCapo-9.12 [5] benchmarks, the largest in the literature on context-sensitive points-to analysis: avrora, luindex, lusearch;
- a number of SPECjvm2008 [44] benchmarks, medium sized computational kernels: compress, mpegaudio, scimark;
- a JavaScript engine implemented in Java;
- the Jolden [10] benchmarks, a port to Java of the pointer intensive Olden benchmarks for C [11];

We analyzed all benchmarks together with the standard Java library and the third party libraries they depend on.

The JavaScript engine benchmark is executed in two contexts. The first scenario is a JavaScript shell, i.e., the engine is built to load and run any script as an input. For the runtime profile extraction we execute it using the delta-blue benchmark. In the second scenario the engine is built as a test framework that executes the ECMAScript test262 [16] JavaScript conformance tests. The test framework is built in Java, hence the analyzed and executed code includes the engine code plus the test framework code in the second scenario.

Since most of the selected dacapo and spec benchmarks or their respective suite harnesses use reflection we patched the code to avoid reflection.

6.2 Profiling Information in the Java HotSpot VM

In order to speculate on the runtime importance of the statically inferred facts we compare those with profiling information collected during the application execution. For this purpose we exploit HotSpot VM profiling infrastructure.

HotSpot VM uses a template based interpreter to execute bytecodes and analyzes the code as it runs to detect the critical hot spots in the program [39, 28]. Additionally, the interpreter collects profiling information including virtual call site receiver types and resolved methods, as well as dynamic type checks (checkcast, instanceof, aastore) receiver types, correlated with execution frequencies. The aastore bytecode stores an object reference into an array of objects verifying that the runtime type of the value is assignment-compatible with the type of the array.

When a method activation counter reaches a threshold, the method is compiled by the dynamic compiler using the profile information. The profile information is used for aggressive optimizations. It can drive dynamic call devirtualization, inlining decisions, checkcast elimination. When the VM shuts down, we dump the collected profiling information for all executed methods.

6.3 Methodology

We begin by analyzing the application and extracting facts about the analyzed code. The facts that we care about are virtual call sites receiver types and resolved methods, and dynamic type checks (checkcast, instanceof, aastore) receiver types.

We then run the application in HotSpot and before the VM shutdown we iterate over the discovered call graph and extract profiling information for the executed bytecodes.

We export the points-to analysis results and the runtime profiling information in files with similar structures. Then we parse the two resulting files for each benchmarked application and compare the results. In this process we do not only collect the number summaries, but also detailed information about the type sets, resolved methods, etc. The detailed information can be used by a developer to inspect the inferred facts and interpret the results in detail.

The focus of this study is to analyze the results of the points-to analysis in the context of hot spots discovered by runtime profiling. To be on par with the runtime profiling results collected by HotSpot VM, which are not context sensitive, we present the results of the context insensitive analysis. Increasing the context depth would improve analysis accuracy, as shown in previous work [49], but the comparison would be less precise.

To reduce the noise introduced by VM calls into the runtime we do not include results for the executed JDK classes. It is not possible to isolate the effect of VM internal calls into the runtime when extracting the profiles. Eliminating the effect of the noise without completely isolating the JDK classes would be possible if runtime profiling was context sensitive.

For both the analysis and the runtime profile we filter the results based on the application and libraries package names:

- dacapo:avrora - avrora, org.dacapo, cck
- dacapo:luindex - luindex, org.dacapo, org.apache.lucene
- dacapo:lusearch - lusearch, org.dacapo, org.apache.lucene
- spec - spec
- jolden - jolden
- jsengine - com.engine.js

6.4 Detailed Results

We present two sets of results. We first compare the size of the static and dynamic discovered call graphs. Then we discuss the projected runtime performance impact of the points-to analysis results, virtual calls and type checks.

6.4.1 Reachable Methods

Table 6.1 compares the number of points-to analysis reachable methods with the number of runtime profile reachable methods.

Because both the points-to analysis and the runtime profiling are context insensitive the statically discovered world is a super set of the runtime discovered world. This fact is hinted in the table by the size of the discovered world.

The last line of the table shows the relative difference between the number of statically and dynamically discovered methods. It is as low as 1.6x for spec:mpegaudio benchmark and as high as 10.3x for jsengine:deltablue benchmark. The reason for the big difference in the jsengine:deltablue benchmark is that the points-to analysis has to be conservative and discover the entire possible reachable world, while the runtime profile is extracted from the execution of a single script which only uses a limited subset of the JavaScript functionality. The difference between the static and dynamic discovered worlds for the jsengine:js262 benchmark is less, 3.9x, because the execution of the js262 conformance tests covers more language features. The same observation applies to the other benchmarks too, the size of the dynamically discovered world is directly dependent on the execution path coverage given by the input data. Choosing a different set of input data would lead to a different path coverage, thus a different static to dynamic discovered world size ratio.

reachable methods #	dacapo			spec			js engine		jolden	
	avrora	lunidex	lusearch	compress	mpega	scimark sor	deltablue	js262	bh	voronoi
static analysis	2286	1126	1143	33	16	25	16807	15772	41	18
dynamic profiling	596	534	316	17	10	12	1626	4061	20	10
static/dynamic	3.8x	2.1x	3.6x	1.9x	1.6x	2.1x	10.3x	3.9x	2.1x	1.8x

Table 6.1: Reachable methods.

In the remainder of the section we only show virtual call and type check data for the methods that were discovered in both points-to analysis and runtime profiling. Being flow insensitive, the points-to analysis reports data for bytecodes that were never executed. We include those for completeness. However, the points-to analysis is able to detect dead code based on the precomputed values of constants or type checks, hence we excluded the respective bytecodes from both the static and runtime profiles.

The numbers are detailed in Table 6.2. The left side presents the virtual calls number summaries while the right side presents the type checks number summaries. The data for each benchmarks is grouped in four execution frequency ranges: never executed, executed less than 100 times, executed more than 100 and less than 9,000 times, and executed more than 9,000 times.

We summarize the important data in Figure 6.1: frequently executed virtual calls, Figure 6.2: virtual calls that are reached by the points-to analysis but never executed, and Figure 6.3: frequently executed type checks.

			virtual calls										type checks				
			static					dynamic					static			dynamic	
		exec #	tbind	1-m	2-m	3-m	p-m	tbind	1-m	2-m	3-m	p-m	folded	decid	!decid	decid	!decid
dacapo	avrora	= 0	366	239	1	8	5	-	-	-	-	-	0	4	8	-	-
		< 100	162	595	10	19	1	153	632	2	0	0	1	9	38	48	0
		< 9k	30	165	6	2	4	30	170	1	0	6	0	2	11	13	0
		> 9k	6	286	1	16	8	6	302	2	2	4	0	0	8	8	0
	lunidex	= 0	818	423	19	3	7	-	-	-	-	-	0	10	18	-	-
		< 100	625	690	40	15	8	570	800	8	0	0	1	28	78	107	0
		< 9k	95	99	22	2	1	71	135	12	1	0	0	1	9	10	0
		> 9k	74	70	2	5	4	65	85	5	0	0	0	3	10	13	0
	lusearch	= 0	414	279	5	0	13	-	-	-	-	-	0	13	22	-	-
		< 100	169	257	13	2	1	169	273	0	0	0	1	6	11	18	0
		< 9k	63	79	8	2	1	62	87	4	0	0	0	1	9	10	0
		> 9k	103	107	19	13	9	94	140	17	0	0	0	1	19	20	0
spec	compress	= 0	3	3	0	0	0	-	-	-	-	-	0	0	0	-	-
		< 100	30	20	0	0	0	30	20	0	0	0	0	0	1	1	0
		< 9k	2	0	0	0	0	2	0	0	0	0	0	0	0	0	0
		> 9k	19	0	0	0	0	19	0	0	0	0	0	0	0	0	0
	mpega	= 0	0	0	2	0	0	-	-	-	-	-	0	0	0	-	-
		< 100	13	9	0	0	0	13	9	0	0	0	0	0	1	1	0
		< 9k	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		> 9k	2	2	0	0	0	2	2	0	0	0	0	1	0	1	0
	scimark sor	= 0	8	3	0	0	0	-	-	-	-	-	0	0	0	-	-
		< 100	11	11	0	0	0	11	11	0	0	0	0	0	2	2	0
		< 9k	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		> 9k	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
js engine	delta blue	= 0	1663	650	34	15	165	-	-	-	-	-	70	46	668	-	-
		< 100	1499	503	85	31	216	1488	782	42	8	14	259	59	1573	1843	48
		< 9k	286	132	58	13	35	284	181	30	0	29	12	30	125	96	71
		> 9k	223	41	37	22	161	207	164	60	19	34	10	10	52	59	13
	js262	= 0	3019	673	28	64	472	-	-	-	-	-	71	30	773	-	-
		< 100	3249	1191	125	61	749	3195	1870	197	63	50	60	100	771	870	61
		< 9k	1614	598	100	48	479	1575	951	116	48	149	35	78	413	407	119
		> 9k	1533	565	124	44	371	1499	810	156	45	127	253	76	1597	1747	179
jolden	bh	= 0	7	3	0	0	0	-	-	-	-	-	0	1	0	-	-
		< 100	29	7	1	0	0	29	8	0	0	0	0	0	0	0	0
		< 9k	8	2	1	0	0	8	3	0	0	0	0	1	0	1	0
		> 9k	53	14	5	0	0	53	15	4	0	0	0	11	1	12	0
	voronoi	= 0	1	12	0	0	0	-	-	-	-	-	0	0	0	-	-
		< 100	16	10	0	0	0	16	10	0	0	0	0	0	0	0	0
		< 9k	0	13	0	0	0	0	13	0	0	0	0	0	0	0	0
		> 9k	0	130	0	0	0	0	130	0	0	0	0	0	0	0	0

Abbreviations

tbind	trivially devirtualizable virtual calls
1-m	monomorphic virtual calls
2-m	bimorphic virtual calls
3-m	trimorphic virtual calls
p-m	polymorphic virtual calls
folded	type checks removed by the preprocessing step
decid	type checks that are decidable
!decid	type checks that are not decidable

Table 6.2: Static vs. Dynamic results.

6.4.2 Virtual Calls

We correlate information from the points-to analysis: number of reachable methods at a given call site, with runtime profile information: number of reachable methods and profiled instruction execution count. We classify the virtual call sites by the degree of morphism. We include numbers for for the invokevirtual and invokeinterface bytecodes.

The first column of Table 6.2, tbind, in both the static and dynamic profiles represents the trivially bindable call sites. These are the call sites that can be devirtualized by a simple class hierarchy inspection, i.e., the resolved method is final or the class that declares it is final. The trivially statically bindable calls are the result of the preceding class hierarchy analysis carried by the Graal compiler, thus we do not count them as points-to analysis results. HotSpot VM profiling also identifies the trivially bindable methods and does not collect detailed receiver type and resolved method information.

The next columns, 1-m, 2-m, 3-m, and p-m classify the virtual call data in monomorphic, bimorphic, trimorphic, and respectively polymorphic virtual call sites. The differentiation between bimorphic, trimorphic, and polymorphic is important: although only the monomorphic call sites can be actually devirtualized, the bimorphic and trimorphic call sites enable more aggressive optimizations, e.g., polymorphic method inlining.

Looking at the summary of frequently executed calls, i.e., more than 9,000 execution counts, Figure 6.1, it is interesting to observe that most of the profiles are dominated by trivially bindable and monomorphic virtual calls. The preprocessing step carried by Graal is able to discover many of the optimizable calls relevant to runtime performance without the help of the points-to analysis.

Looking at the dacapo benchmarks we observe that the points-to analysis is able to discover a high percentage of the virtual call sites as monomorphic, very close to the accuracy of

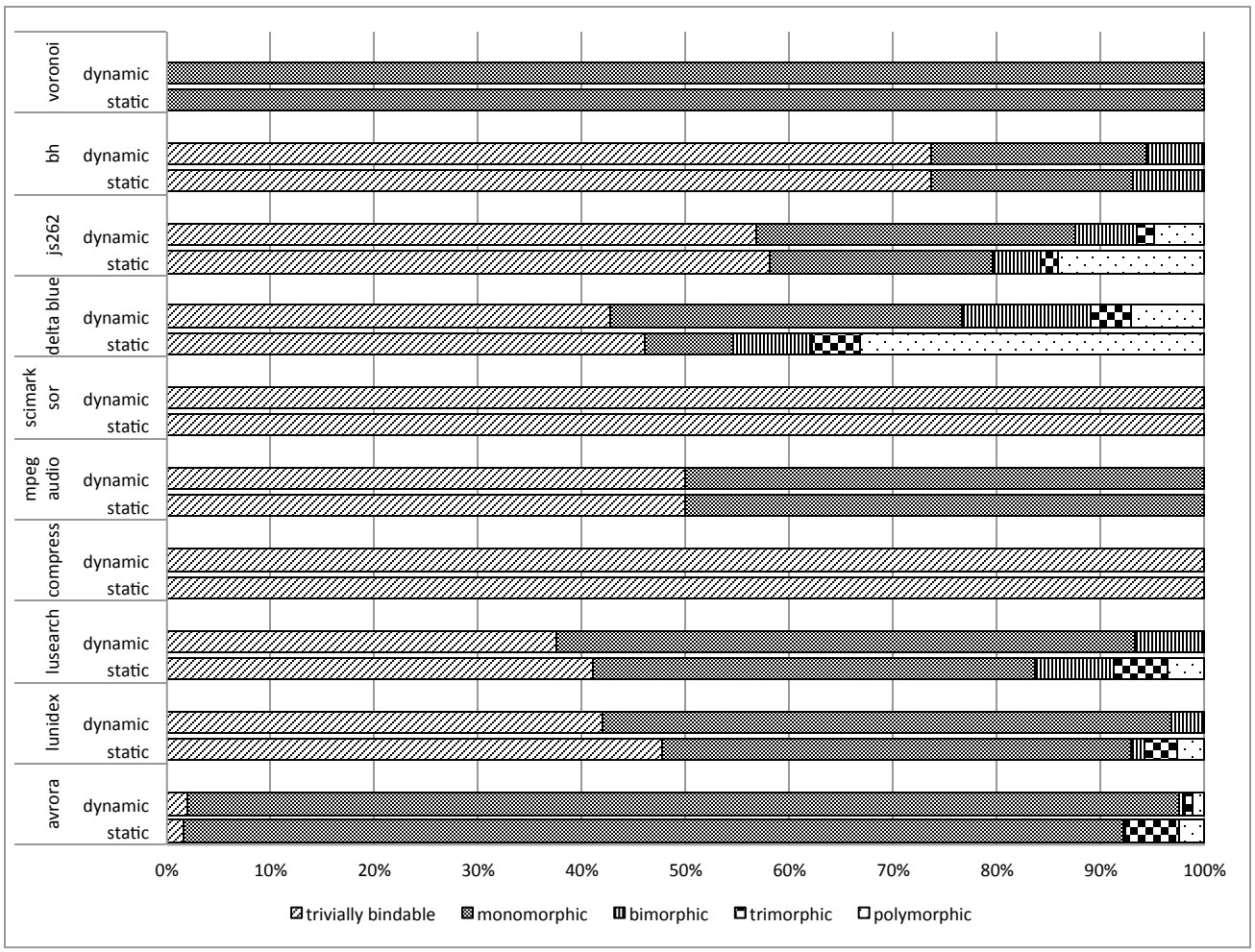

Figure 6.1: Frequently executed virtual calls, more than 9,000 times

the runtime profile. Correlating this with the fact that the majority of the virtual calls are executed less than 9,000 times, for this set of benchmarks increasing the accuracy of the points-to analysis, i.e., inferring that more call sites are monomorphic, would have a marginal impact on the performance of the compiled code. It is also interesting to note that the number of trivially devirtualizable call sites is an important fraction of the total number of virtual calls for the luindex and lusearch benchmarks. Therefore we find that the preprocessing step of identifying trivially devirtualizable call sites is essential in reducing the complexity of the analysis.

Next we inspect a set of medium sized computational kernels extracted from the spec suite. The common characteristic of these benchmarks is that they are small but computationally intensive. Their use of object-oriented features is minimal and as a consequence they do not exhibit a high number of virtual calls. Hence the points-to analysis discovered facts about the virtual calls are as accurate as those discovered by runtime profiling.

Our JavaScript engine is an interesting benchmark for the study of pointer analysis implementations. It represents the programs internally as an abstract syntax tree (AST) and it relies heavily on the use of virtual call dispatch to select the correct implementation of an operation. Interestingly the majority of the virtual calls are trivially devirtualizable for both engine configurations, delta-blue and js262. However, given the complexity of the analyzed class hierarchy, the context insensitive analysis does a poor job at inferring that a number of frequently executed virtual calls are monomorphic, almost as much as 4 times virtual calls are counted by the points-to analysis as being polymorphic compared to the runtime profile. Yet the accuracy in discovering bimorphic and trimorphic call sites is comparable to that of the runtime profile. Enabling context sensitivity for this set of benchmarks would improve the accuracy of the results.

The next set of benchmarks, jolden, are a Java port of the C language pointer intensive Olden benchmarks. As in the case of the spec benchmarks the points-to analysis is able to create an accurate image of the runtime profile. This is due to a limited amount of polymorphism. Looking at the voronoi benchmark we note that the majority of the virtual calls are frequently executed, the consequence of virtual calls in long running kernels.

Previous work on points-to analysis classifies virtual calls in monomorphic and polymorphic without distinguishing between trivially devirtualizable calls. We believe that distinguishing trivially devirtualizable calls is important to give a more accurate understanding of the real impact of points-to analysis.

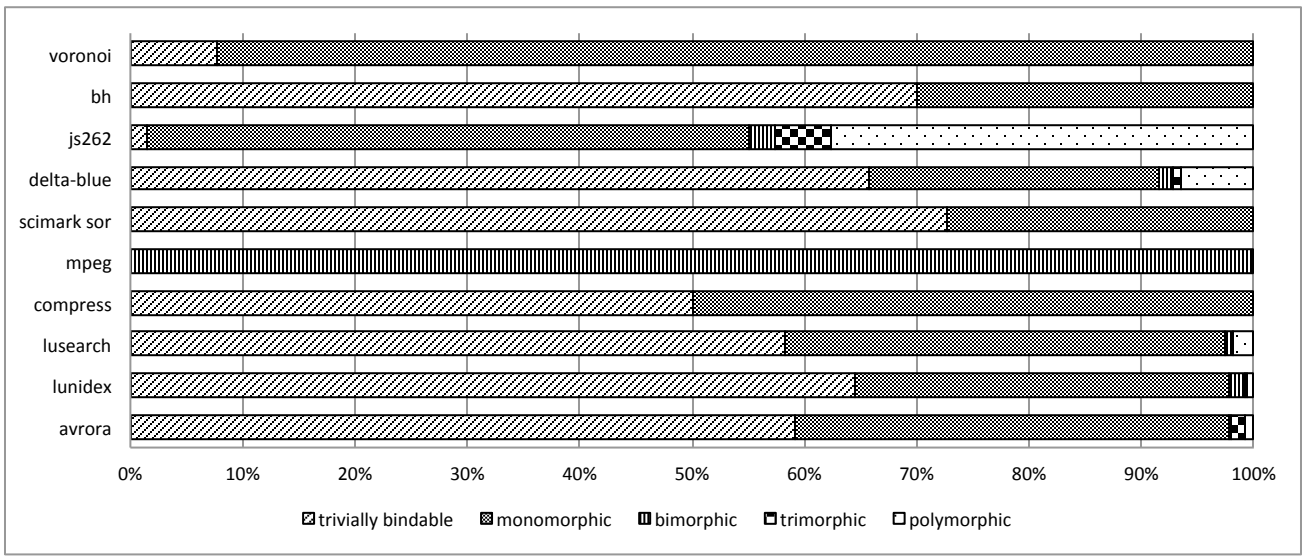

Figure 6.2: Never executed virtual calls

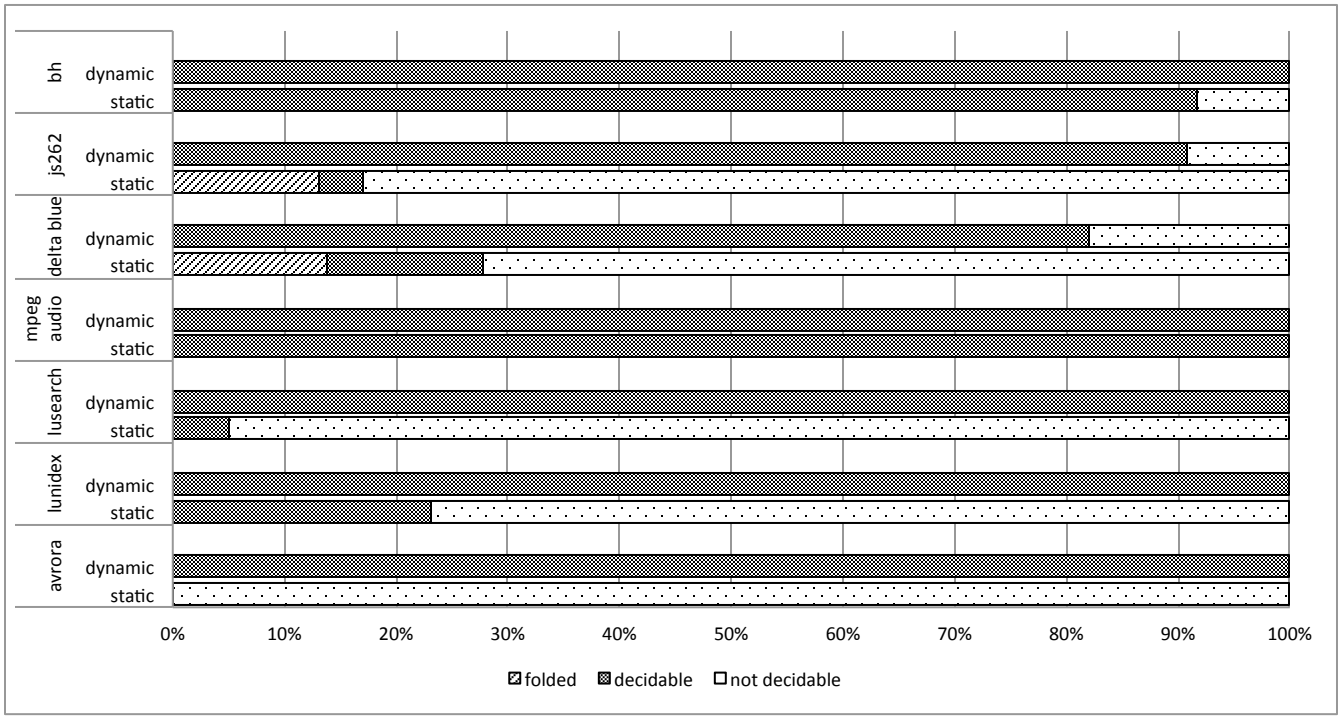

Figure 6.3: Frequently executed type checks, more than 9,000 times

6.4.3 Type Checks

The right half of Table 6.2 presents the numbers found for type checks, checkcast and instanceof bytecodes. The facts inferred by the points-to analysis, type sets recorded at a

given type check, are correlated with runtime profile information, type check instructions execution count and receiver types. We divide the type checks in decidable and not decidable according to the recorded receiver types. A type check is decidable if it either passes or fails for all of the recorded types. Proving that a type check instruction is decidable enables more aggressive compiler optimizations.

Since the input of our analysis is Graal IR we present in a separate column the type checks that are removed by the canonicalization preprocessing step. The canonicalization step can decide to remove type checks using a simple inspection of the type check receiver object declared type and of the condition type. For example for an instanceof if the receiver object declared type is a subtype of the condition type and the receiver object cannot be null then the type check can simply be removed. For checkcast the same rule applies with the amendment that the receiver object can be null.

Figure 6.3 presents the distribution according to decidability of the frequently executed type checks. Some benchmarks do not have any frequently executed type checks, hence they are removed from the graph. Only for the jsengine benchmarks a significant number of frequently executed type checks are eliminated by the class hierarchy analysis carried by Graal.

Overall we observe that the dynamic profile is more accurate than the points-to analysis with respect to type checks. The number of type check instructions that are decidable at the end of the profiling, and that enables JIT optimizations, is higher than the number of type checks that are guaranteed as removable by the points-to analysis.

For the selected dacapo benchmarks most of the type checks have a low execution count and the preprocessing step only removes a single type check. The spec benchmarks have few type checks too.

For the jsengine benchmarks the preprocessing removes a good number of type checks, many of them with a high execution count. However, the discrepancy between static and dynamic

profile is even more pronounced. The points-to analysis infers that only a small number of type checks can be removed in comparison to the number of type checks that the runtime profile finds decidable. The projected impact of this inaccuracy is significant since most of the checkcast have a high execution count, above 9,000.

Previous work classifies type checks in *may* or *may not* fail. Type check static decidability is a super set of the type checks that may not fail, including those that always fail, and enables JIT compilers to do more aggressive optimizations. Previous work does not mention easily removable type checks.

6.5 Conclusions

This study emerged from our own experience with implementing a points-to analysis phase in our compilation pipeline and discusses how we think that the benefits of a points-to analysis step can be maximized. This work is a necessary first step to reach our goal of efficiently compiling Java ahead of time. We believe that switching from a whole-program analysis to a demand-driven analysis that exploits runtime profiling information is the key to achieve better runtime performance while keeping the analysis costs under control.

We presented a comparison between points-to analysis and runtime profiling and identified the differences and similarities. We hope that this study can stand as a guidance to JIT developers that could utilize these insights for better optimizations.

Chapter 7

Memory Region Aware Points-to Analysis

In this chapter we present in detail the memory region aware points-to analysis. We begin by introducing the analysis formulation and discussing how it extends the points-to analysis. Then we examine optimizations that enhance efficient allocation and garbage collection while preserving memory safety.

7.1 Analysis Formulation

The region analysis uses the programmer-defined regions as additional context information. By augmenting the context with region elements the points-to analysis can disambiguate the various abstractions of objects allocated in multiple regions. A method is analyzed separately for each different region from which it is invoked. Each allocation site has a different abstraction for each region in which it is encountered.

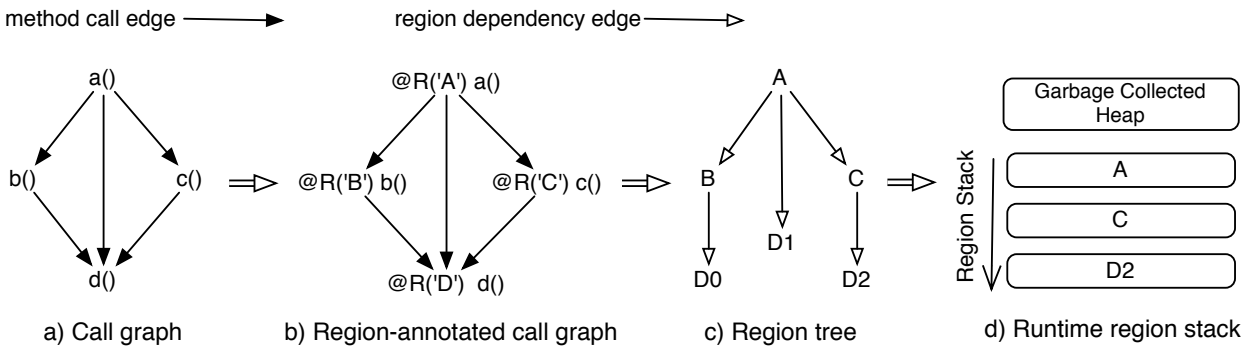

Figure 7.1: Call graph transformation.

To keep track of the region context, the analysis builds a region tree, as depicted in Figure 7.1. It does this on the fly while discovering the call graph: each region-annotated method adds a new node in the region tree when it is analyzed. The graph formed by the region-annotated methods is transformed into a tree by cloning each region that has more than one parent. Transforming the call graph into a region tree instead of a region graph simplifies the implementation, because a tree node implicitly stores its context in the path to the root. During this transformation the analysis eliminates cycles created by across-region recursion as discussed in Section 7.2.

At runtime, the memory regions form a stack. The various paths through the tree correspond to runtime region stacks. When talking about a region's age relative to its position in the region stack a region closer to the bottom of the stack is older than a region closer to the top of the stack. Thus, an older region has a larger scope than a younger region.

By keeping track of the region context, the analysis extends the call-stack context. Thus each method, in addition to the calling context, is analyzed in the region context of its caller or, if it is region annotated, it expands the region tree and is analyzed in the newly created region. Each object has a different abstraction for each context in which it is discovered, hence for each different region from where its allocator method is invoked. In this new formulation a context-sensitive analysis configuration such as *2-object-sensitive with 1-context-sensitive heap analysis*, i.e., a calling context of depth 2 with a heap context of depth 1, becomes *2-*

object-sensitive with 1-context-sensitive heap and 1-region-context, where the added element represents the active region when the current method is processed.

7.1.1 Analysis Results

To facilitate the discussion we first define a *region mapping function* that maps each allocation site to a set of tuples *(definition region, allocation region, offset)*, shown in Figure 7.2.

$m(A) = \{(d, a, o)\}$, where
m is the region mapping function,
A is an allocation site,
d is definition region context,
a is allocation region,
o is offset in runtime region stack

Figure 7.2: Region mapping function.

The analysis determines for each abstract object a *definition region* to *use region* mapping. The *definition region* represents the active region context when the abstract object is created, i.e., its allocating method is analyzed. The *use region* represents the lowest common ancestor, relative to the region tree, of all region contexts in which the abstract object is used. Object uses are field stores, field loads, invocations and reference comparisons. Return statements cause objects to escape to the caller's region. Field stores to static fields cause objects to escape all regions and be allocated into the garbage-collected heap. Due to context sensitivity, an allocation site generates as many abstract objects as the contexts in which its allocator method is analyzed. Consequently, when the analysis converges each allocation site is characterized by a set of definition-region-to-use-region mappings, one mapping for each region tree branch in which the allocation site is analyzed.

For each allocation site each *use region* is, in runtime terms, a possible *allocation region*. At runtime, the decision of which specific *allocation region* to be used at a particular moment

is based on the runtime region context. The runtime region context is given by the region at the top of the runtime region stack when the allocation instruction is executed. The top of the runtime region stack is guaranteed by the analysis to be among the allocation site's definition regions. Thus, the runtime system chooses the *allocation region* corresponding to the runtime region stack top in the allocation site's analysis results. To facilitate efficient allocation, we attach to the (*definition region*, *allocation region*) pair a new value, an *offset*. The *offset* represents the distance of the *allocation region* from the top of the runtime region stack, i.e., the *definition region*. The *offset* value is a compile-time constant. It is used to efficiently determine the allocation region based on the runtime region context. Thus, the runtime system does not have to query the runtime region stack looking for the position of the allocation region. Furthermore using this strategy, a memory management system using our analysis does not need to pass regions as parameters to functions. The details of how the offset is used for an efficient runtime region management are discussed in Chapter 8.

7.1.2 Region Mapping Examples

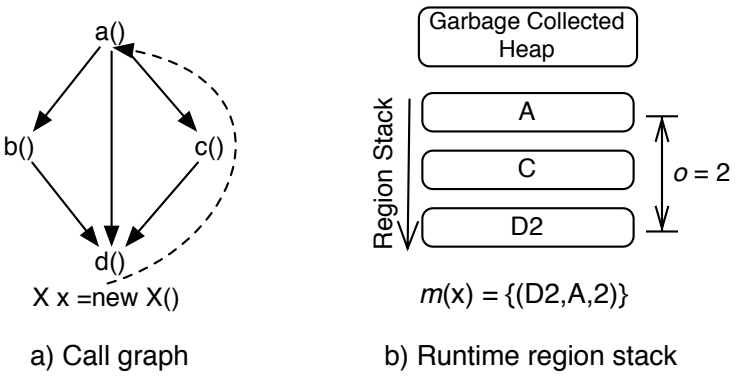

Figure 7.3: Allocation *offset* example.

In this subsection we present two examples of region allocation results. The first example, Figure 7.3, is used to demonstrate the use of the allocation *offset*. The second example, Figure 7.4, is more complex and exemplifies the region mapping for an object escaping on multiple paths.

Figure 7.3(a) shows the call graph discovered by the static analysis, the same as in Figure 7.1. For simplicity, we assume that all the methods in the presented graph are region annotated. On one of the call paths the object x allocated in region D escapes to region A. Figure 7.3(b) shows the state of the runtime region stack when the allocation code for object x is executed. At runtime when object x is allocated it's definition region, *D2*, is at the top of the runtime region stack. The effective allocation region for object x is region A which is at an offset of *2* from region *D2*. Thus, the mapping for object x is *(D2,A,2)*. The offset is a compile time constant and it is used to efficiently find the allocation region given the current region context, i.e., a definition region, for object x.

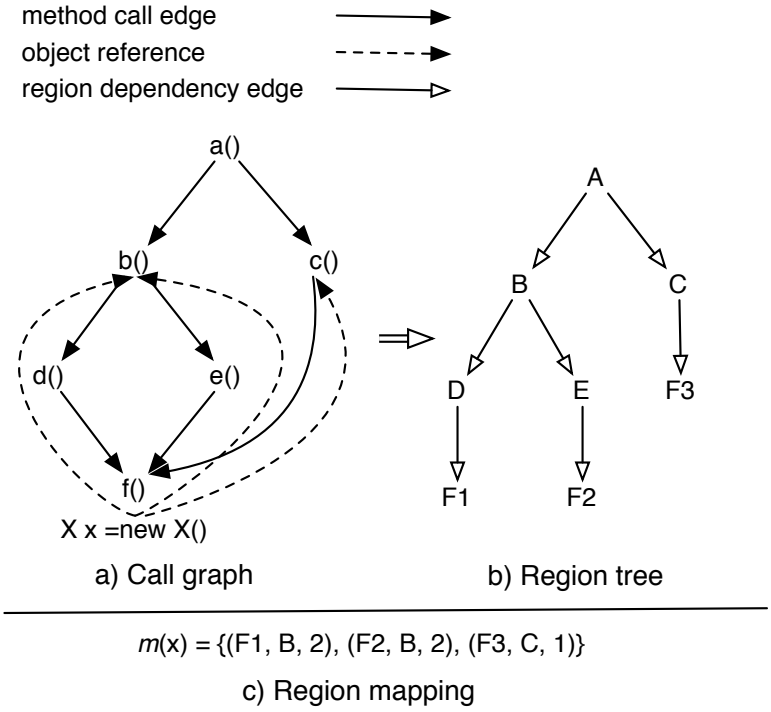

Figure 7.4: Region mapping example.

Figure 7.4(a) shows the region-annotated call graph for a short program. In the example there is a single allocation site in method f(). The x objects allocated in the region defined by the method f() may escape on two separate paths to the region defined by the method b() and on a third path to the region defined by the method c().

The corresponding region tree is presented in Figure 7.4(b). Region F is cloned three times, once for each separate region context that reaches it.

Figure 7.4(c) shows the results of the region analysis. The x objects allocated in f() are mapped to three different regions depending on the active region context.

7.2 Recursion

We distinguish between in-region recursion and across-region recursion. In-region recursion is defined as recursive calls of methods that do not open a new region. The non-escaping allocation sites are mapped to the last opened region, thus it does not affect the region analysis.

Across-region recursion creates cycles containing one or more regions and is generated by recursive calls across region declaring methods. We treat across-region recursion by not reentering a region if it is already present in the region tree on the path from root. There is only one copy of each region on each distinct tree path. This is a conservative solution which preserves the correctness of the analysis. Reentrant region-annotated methods map non-escaping objects to the region opened when the method was first entered. Alternatively the analysis could allow a limited size recursive tail of regions. This approach would divide the total allocated memory across the reentrant region-annotated method in smaller regions. However, when the recursion tail limit is reached, objects would have to be conservatively allocated in the last opened region corresponding to the reentrant region-annotated method.

Across-region recursion requires a runtime check every time when a region-annotated method is called. If the annotation declares a region that is already in the region stack the state of the stack does not change. Otherwise, a new region is pushed onto the stack.

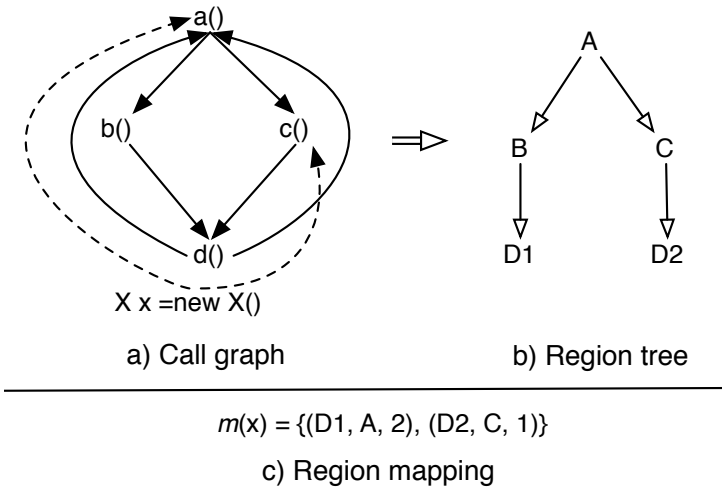

Figure 7.5: Region analysis recursion example.

Figure 7.5 presents an example of across region recursion. Assuming again that all methods in the presented call graph are region annotated, all declared regions are reentrant. In Figure 7.5(a) on one call path object x escapes up to the region defined by the call to *a()* and on the other call path it escapes up to the region defined by the call to *c()*. On both call paths the shape of the call graph generates region cycles. This makes it impossible to statically map each allocation site to the region at the right recursion depth. Thus, it conservatively builds a simplified region tree that has a single instance of the reentrant regions for each path from root, as depicted in Figure 7.5(b). Using the simplified region tree, finding the region mapping is straightforward and the result is presented in Figure 7.5(c).

7.3 Safety Preserving Invariant

The described region analysis allows both pointers from older regions to newer regions and from newer regions to older regions. The pointers from older regions to newer regions can turn into dangling pointers when the newer region is deallocated. Consider the example in Figure 7.6. In Figure 7.6(a), an object from a newer region pointing to an object from an older region is safe and it cannot result in a dangling pointer. However, in Figure 7.6(b) when

region B is deallocated then x.f becomes a dangling pointer. If x.f is not dereferenced in the context of region A (this can be guaranteed by the static analysis) this dangling pointer is safe at runtime. Yet, the region analysis must accommodate tracing pointers when garbage collection is triggered. Thus we insert an additional analysis step that prevents dangling pointers.

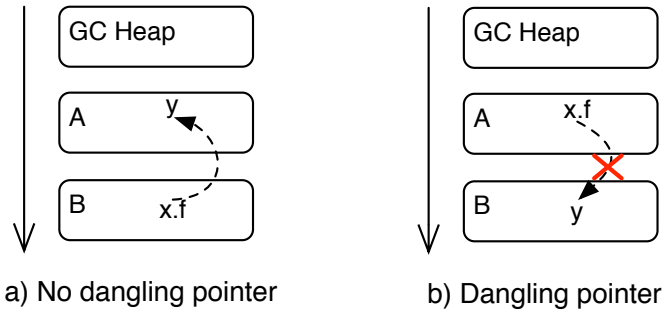

Figure 7.6: Region analysis invariant.

The additional step processes the abstract objects using a fixed-point approach. It enforces the invariant that the allocation region of an object cannot be older than the allocation regions of the objects that it references, it must be at least the same age or younger. A referenced object that breaks this invariant is hoisted to the referencing object's allocation region and, since its allocation region changed, the analysis processes the objects that it may further reference. In Figure 7.6(b), to respect the invariant, object y must be hoisted to region A.

Following the same approach as in previous example Figure 7.7 shows how the region analysis modifies the region mappings to enforce this invariant. Initially, the y objects allocated in d() escape to region B and C. On the call path from c() objects returned by d() are assigned to an object x that escapes to the region context A. To prevent x.f to become a dangling pointer, the analysis hoists y into the same allocation region as x. Out of the two y abstract objects only the one in region context D1 is updated.

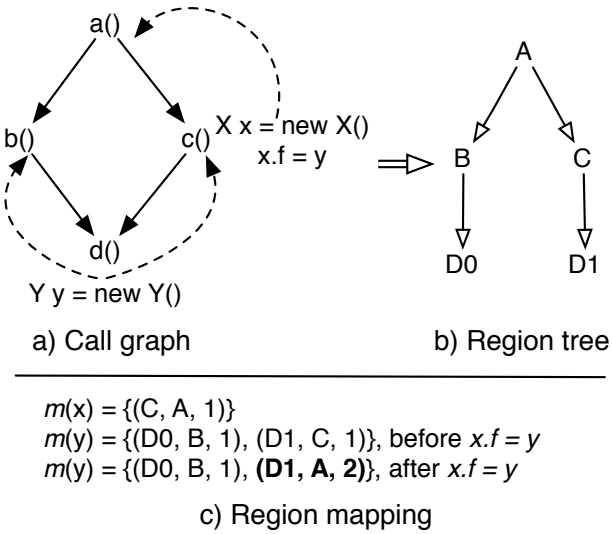

Figure 7.7: Region analysis invariant example.

The problem of dangling pointers could also be solved through a runtime mechanism, avoiding weakening of the region analysis. A write barrier could keep track of all the pointers inside a region that come from an older region. Then, when the younger region is deallocated all the resulting dangling pointers could be invalidated. However, this approach would increase the runtime overhead.

7.4 Efficient Allocation

The runtime system must keep track of the region stack. The regions on the stack are scoped, i.e., a region at the bottom of the stack has a wider scope than a region closer to the top. When a region is entered its identifier is pushed on the stack; when the region exits its identifier is popped off the stack. At the bottom of the region stack is, conceptually, the garbage-collected heap. This represents the memory region that is opened when the application starts and is closed when the application terminates.

Our analysis does not require passing region descriptors as parameters to methods at runtime. A straightforward way to determine the allocation region for an allocation site would be to query the analysis results for the allocation site and determine the corresponding allocation region given the current active region. The current active region is always at the top of the region stack and it corresponds to a definition region in the analysis results. Then, the allocation region descriptor would be used to lookup the runtime region stack for the right allocation space. However, this would incur a high overhead in both the space needed to store the analysis results for each allocation site, and in the time required to query the mapping and to lookup the runtime region stack.

The first step in optimizing the region allocation is reducing the analysis results to an *allocation site region mapping table*. As discussed in Section 7.1.1, we compute an *offset* in the runtime region stack for each region mapping . Using the *offset* the allocation region can be easily discovered by offset arithmetic, thus avoiding the runtime region stack lookup. To find the correct offset given the active region context, we assign each region a unique ID. Thus the allocation site region mapping table consists of rows of region ID to allocation offset mappings Since the number of regions is small the offset table size is manageable. We call this strategy *hybrid allocation table* in the evaluation.

A table lookup for each object allocation still incurs a significant overhead. To further reduce the allocation overhead we can trade-off some of the analysis precision. We modified the analysis to normalize the region mappings such that each allocation site has a unique runtime offset. The offset is a compile-time constant, unique for each allocation site, and can be injected in the allocation code. The unique offset is the maximum of all offsets of an allocation site's mappings. Using this approach the object is not always allocated in the optimal region, i.e., the youngest possible region in the stack, but in an older region. This optimization affects the allocation sites that have a high variance among their allocation

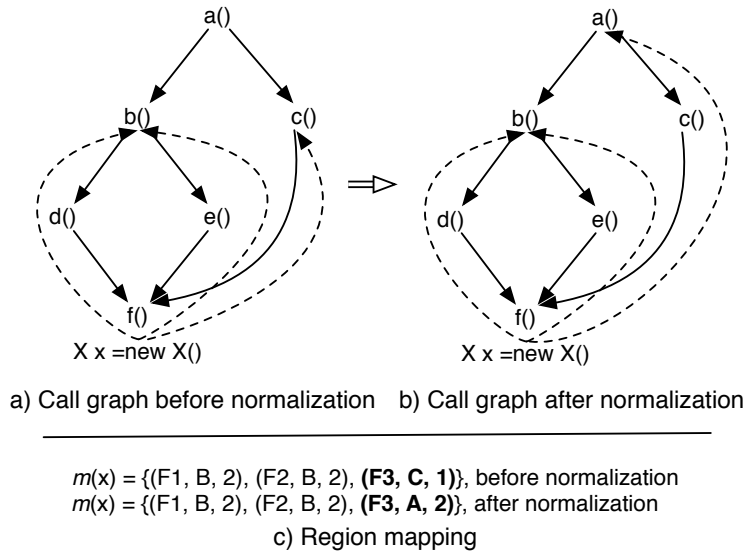

Figure 7.8: Region analysis normalization example.

offsets. In the worst case the unique offset is higher than the stack depth on some paths in the region tree. In this case we conservatively allocate into the garbage-collected heap.

The region analysis preserves correctness using a fixed-point approach. Transformations that break the safety invariant cause a chain of updates that restore the invariant, e.g., moving an object into an older region through offset normalization effects in updating the region mappings of all reachable objects. The algorithm reaches a fixed point when all the mappings respect the safety invariant.

In Figure 7.8 we continue the example in Figure 7.4 and show how the offset normalization affects allocation. When reached in context F3, object x is initially allocated in region C at offset 1. After normalization the unique offset of x's allocation site has a value of 2, thus, when reached in context F3 object x is allocated in region A. The unique offset can now be used by the allocation code for x at compile time. This transformation only impacts the allocation on one of the region paths, the object is allocated in an older region than the optimal one. We call this strategy *hybrid normalized* in the evaluation.

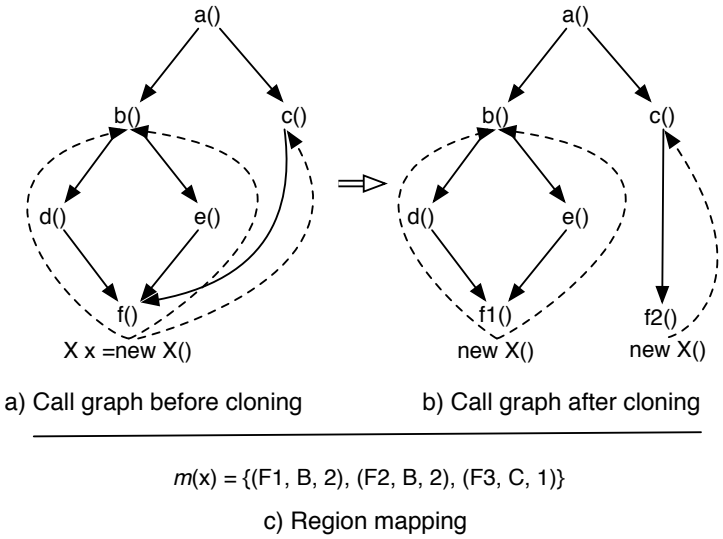

Figure 7.9: Region analysis cloning example.

Another approach for performing allocations without a table lookup would be method cloning. By cloning methods at compile time for each region context in which they are reached the allocation site mappings are split and result in unique, compile time constant offset values. In Figure 7.9 we reuse the example from Figure 7.4 and show how method cloning transforms the call graph. In the worst case method f() is cloned three times, since it is reached from three different region contexts. However, in this example the mappings of the allocation site in f() have only two different offset values. This results in only two versions of f(). Therefore by analyzing the region mapping offsets of the various allocation sites belonging to a method and identifying the conflicts, i.e., those mappings that have different offset values, the cloning factor can be reduced. Method cloning has unwanted side effects, such as increasing the code size of the application. Our evaluation shows that the normalization based approach works well in practice, so we did not implement method cloning.

7.5 Efficient Garbage Collection

To support unbounded regions that can increases dynamically to accommodate object allocation, the region memory must be organized in chunks. The memory system can use either fixed or variable size chunks. The chunks corresponding to a region are linked together and when the last chunk is full, a new one is requested from the runtime system. Each region is managed by a region descriptor that keeps track of the linked list of chunks. To enable fast region allocation, the region descriptors must track the last free location in the last chunk.

When a region exits, its chunks are returned to the runtime system without the need for garbage collection. However, the implementation must be conservative and assume that the size of the allocated memory can increase beyond the reserved heap space. This triggers a garbage collection. In general any garbage collection algorithm can be adapted to function within a hybrid memory management scheme. Using a generational GC the garbage-collected heap is divided into old and young generations. In this scenario the region-allocated memory is conceptually part of the young generation and must be scavenged together with it.

To maintain the region safety invariant, region-allocated objects surviving GC must be placed into the same regions they were allocated into. This requires separate survivor spaces for all regions. To simplify the collection, we decided to break the region invariant during GC and copy all surviving objects into the garbage-collected heap. All regions active at the time of GC must be marked as inactive, i.e., the region stack is not cleared per se, it is *logically* reset to the empty stack state. The *physical* shape of the region stack must always correspond to paths from the root in the region tree, even when GC occurs. This is required by the region reentrancy check which detects across-region recursion. Attempting to allocate into an inactive region forces conservative allocation into the garbage-collected heap and exiting an inactive region does not deallocate any memory. Regions are reactivated the next time they

are entered, resuming normal region allocation. This approach does not need any changes to the GC algorithm.

The region stack implicitly orders objects by their lifetime. This would allow a partial GC when the size of a region exceeds a configurable allocation threshold. The partial GC only needs to scavenge the region that exceeds the threshold together with the younger regions. The region safety invariant ensures that there are no outside references into the scavenged space since none of the older regions can have pointers into the younger regions. In our experiments, this approach is not necessary however since the maximum size of a region stays well below every reasonable region GC threshold.

Chapter 8

Hybrid Memory Management in Substrate VM

In this chapter we present details of the region analysis implementation and discuss how the memory management system was modified for an efficient hybrid memory management using the results of our analysis.

8.1 Region Analysis Operation

We implemented the region analysis in an ahead-of-time (AOT) compilation system for Java. The system assumes a closed-world of Java code, so that all reachable code can be determined by the static analysis before the AOT compilation. The Graal compiler [37] is used both for the static analysis and the AOT compilation. The resulting executable contains application code, third party libraries, the reachable parts from the Java class library, as well as a Java runtime system including a generational GC. The memory region analysis uses the static

analysis results computed for AOT compilation, i.e., we do not perform a separate static analysis just for the memory region analysis.

However, our approach is not limited to AOT compilation. The analysis can be integrated in a traditional Java VM that performs just-in-time (JIT) compilation of frequently executed methods. The static analysis can run during the warm-up period of the application, so that the region information is available to the JIT compiler when it optimizes a method. Or the runtime system can perform the pointer analysis online during program execution as proposed by Hirzel et al. [21]. Because of the explicitly marked memory regions, the static analysis does not need to analyze all Java code (e.g., the whole JDK). This eliminates the requirement of a closed-world assumption for the static analysis. At the same time, this eliminates the scalability and precision limitations that a whole-program points-to analysis has when analyzing a language like Java.

The region analysis is built on top of the core points-to analysis. The abstraction of heap objects is extended with a definition region. Likewise, the method abstraction is extended with a region context. This allows the analysis to maintain separate points-to sets for each region context. The region analysis operates in a similar manner with the core points-to analysis. Each allocation site has attached a region state. The region state is a static representation of the region mappings, i.e., it maps definition regions to use regions and keeps track of the region offset for each mapping. A region flow is used to model region dependency between allocation sites, i.e., keep track of objects that can be referenced at runtime. The region dependencies are generated by field stores. When the region state of an allocation site is updated, the state of all the dependent allocation sites is updated to ensure the region safety invariant. The region analysis is implemented as a fixed point algorithm and it ends when all the region mappings are stable, i.e., the region dependencies don't cause any region mapping updates.

8.2 Heap Organization

We built the hybrid memory management scheme on top of the generational GC. A high level view of the heap structure is presented in Figure 8.1

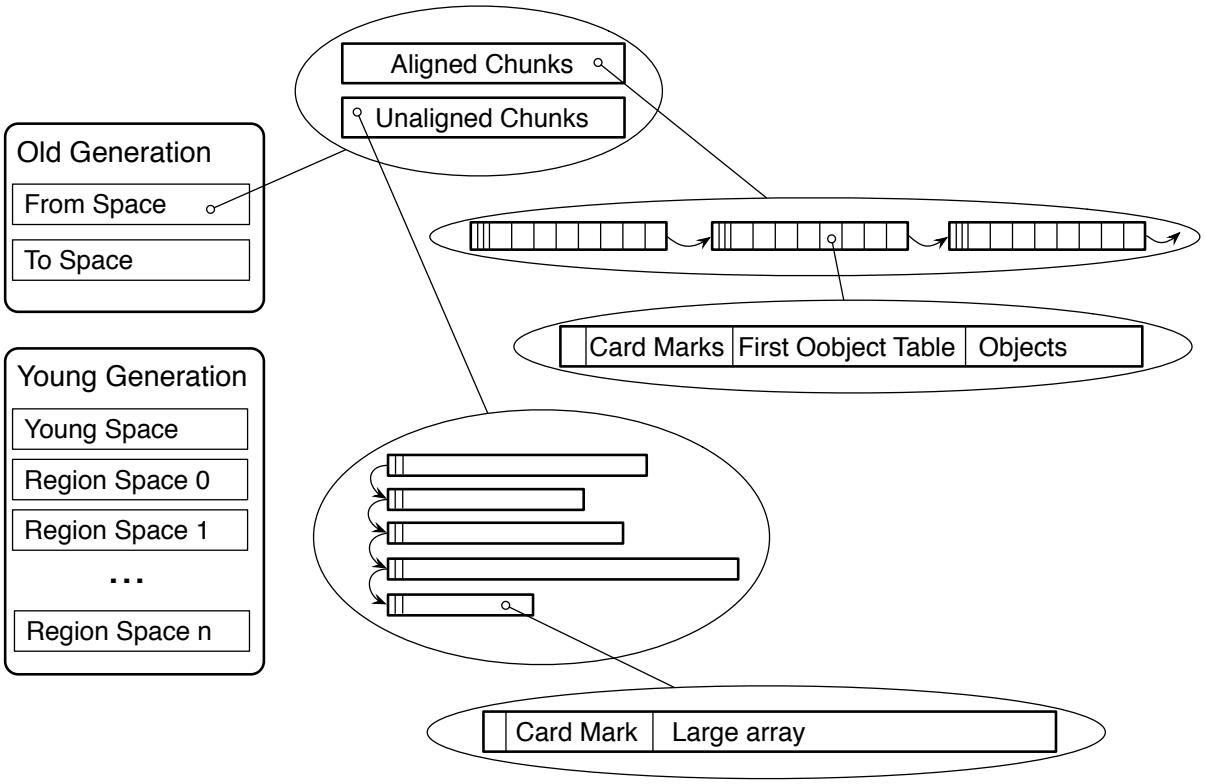

Figure 8.1: Heap organization.

The garbage-collected heap consists of two generations, young and old [55]. We use a stop-and-copy algorithm for both generations. The incremental GC copies all live objects from the young generation into the old generation, i.e., there is no aging of objects in the young generation. The full GC copies all live objects from both generations into the survivor space of the old generation. The region stack is conceptually part of the young generation and is scavenged together with it when the allocation threshold is reached.

A space is the building block of a region. The old generation has two spaces, *from* and *to*. During a full GC *to* is the survivor space, i.e., the destination space of all live objects in the

young generation and the old *from* space. At the end of a full GC the two old spaces are swapped and all the live objects are in the *from* space. In the normal GC scheme the young generation consists of a single space. This is the space where objects are allocated. In the hybrid memory management scheme the young generation is extended with a separate space for each region. The original young generation space is used for conservative allocation, i.e., for objects that cannot be region allocated. The maximum number of region spaces, i.e. the upper bound of the runtime region stack size, is known statically. Thus, the region spaces act as containers for the region-allocated objects. When the region exits the memory corresponding to the space is simply released.

A space must grow flexibly to accommodate for allocated or moved objects. That is why spaces are organized in chunks [25, p. 12]. There are two types of chunks: *aligned*, fixed size and *unaligned*, variable size. The *aligned* chunks are aligned on size-granularity and are used for small object allocation The *unaligned* chunks are used for large array allocation and are never moved. Each space keeps a linked list for each type of chunks. On the fast path the runtime system tries to allocate objects in the last aligned chunk. If the allocation succeeds, i.e., the object fits in an aligned chunk and the last chunk has enough free space, the allocation is as simple as bumping a pointer and returning the start address for the allocation. If the object fits in an aligned chunk but the fast path allocation fails because there is not enough free space in the last aligned chunk then a new aligned chunk is requested. Aligned chunks are requested from a pool of free aligned chunks. If the free aligned chunks pool is empty then the runtime system allocates memory from the operating system by reserving a block of virtual address space at the chunk alignment and of the chunk size. If the fast path allocation fails because the object is a large array then the runtime system reserves a block of virtual address space of the required size at any alignment. When the memory corresponding to a space is released the aligned chunks are returned to the pool of free aligned chunks. If the pool is full the aligned chunks are returned to the operating system, i.e., their memory is freed. The unaligned chunks are always freed.

8.2.1 Card Remembered Sets

Substrate VM uses card remembered sets [26, p. 171] for efficient garbage collection. A potentially time consuming part of the incremental GC is finding all live objects in the young generation. The naive approach would be to scan the entire old generation to discover pointers in the young generation. A better solution is keeping track of the old generation segments where pointers to objects have been stored. By doing so only those segments that are known to point to objects in the young generation must be scanned.

The first step in achieving this is breaking down memory chunks into segments called cards. Each chunk reserves some space for the card marking table, a remembered set that summarizes pointer stores into a region. Each card has a 1 byte entry in the card marking table. A card is *dirty* if a pointer has been stored recently into the memory summarized by the card, or *clean* otherwise. Thus, when looking for roots into the young space, the whole old space need not be searched for pointers, only the parts of the old space covered by dirty cards.

At each pointer store the card corresponding to the destination of the store is dirtied. At each collection, the dirty cards are scanned and the corresponding memory examined for pointers to the young space. When the memory has been scanned, the corresponding card is cleaned.

Sine cards divide the chunks into fixed segments it is possible that some objects cross cards boundaries. A *first object table* is used to store the start of the object that crosses onto the memory covered by a card. Once the header for the first object in a segment is found the objects are scanned by striding with the size of the object.

8.3 Region Manager

The region manager is a runtime component. When a region-annotated method is called, the region manager pushes a new region on the runtime region stack. When the method returns, the region manager pops the region and releases its memory.

The *enter region* operation first checks if a region of the same *id*, i.e., generated by the same annotation, is already on the region stack. If there is such a region then there is an across region recursion situation and no new region is pushed onto the stack (see Section 7.2). Objects are conservatively allocated in the existing region. The *exit region* operation just pops the top region.

The operation of the region manager is dictated by snippets of code inserted in the method graphs by the region analysis. The region analysis extends the Graal graphs with *region enter* and *region exit* nodes. These nodes are lowered to snippets in the later phases of compilation. Both snippets are essentially a runtime call to the region manager for the appropriate operation. The *enter region* snippet passes in the *id* of the new region. The new region *id* is used by the region reentrancy check.

The region manager also keeps track of the region context, i.e., the region at the top of the runtime region stack. The region context is used for querying the region mappings for the allocation region offset when the analysis results are not normalized.

8.4 Region Allocation

Region allocation is achieved by modifying the Substrate VM allocation mechanism. Substrate VM object allocation mechanism is built with Graal snippets. The static analysis creates SVM specific allocation nodes that hold information about the type and size (in case

of arrays) of the objects. The nodes are lowered to code snippets that make calls into the runtime system for reserving and formatting the memory for the new objects. On the fast path these calls are inlined and thus there is no overhead.

The region allocation extends the normal allocation mechanism in that it first finds the matching region space instead of allocating directly to the young generation space. The region analysis extends the allocation nodes with the region mapping metadata. When the region offsets are normalized the metadata is summarized to a compile time constant integer, i.e., the allocation offset (see Section 7.4). When the region offsets are not normalized the region mapping is queried to find the right allocation offset. The current region context, provided by the region manager, is used as a key in the region mapping lookup. Using the allocation offset the runtime finds the allocation region space by indexing in the runtime region stack. After the right region space is found allocation works as before.

8.5 Region-Allocated Objects Collection

The region stack organization is transparent to the GC. From the collection algorithm's perspective the region stack is just a part of the young generation. The live objects are discovered just like before, by scanning from roots (old generation and call stack). The region-allocated live objects are conservatively promoted to the survivor space. To ensure correctness, the garbage-collected regions are marked as inactive and reactivated next time they are entered, as discussed in Section 7.5.

The alternative to this would be having a separate survivor space for each region. This could be achieved by adding an extra space to each region, following the *from-to* space old generation organization model. Then the GC would have to promote live objects to the corresponding region surviving space. The only additional requirement would be fast

retrieval of the region in which the object is allocated. This is easy for unaligned chunks, the head of the unaligned chunk is at a fixed offset from the object header. For objects in aligned chunks the header of the chunk is at a chunk size granularity and each chunk keeps the address of its space.

Chapter 9

Hybrid Memory Allocation Evaluation

In this chapter we evaluate the accuracy of the region analysis and the efficiency of the hybrid memory management technique.

To validate the technique we use the Jolden [10] benchmarks, a port to Java of the pointer intensive Olden benchmarks for C [11]. We choose those JOlden benchmarks that exhibit transactional execution patterns and are best suited for our technique. The omitted benchmarks allocate a big data structure (a tree or a graph) on which the algorithm is run without too much phase local allocated memory.

To test scalability we use SPECjbb2005 [50], a memory intensive benchmark whose execution pattern fits the intended use of our technique. SPECjbb2005 evaluates the performance of server side Java by emulating a three-tier client/server system, with emphasis on the middle tier. It makes heavy use of dynamic memory allocation, one of its design goals being to exercise the implementation of garbage collectors. By analyzing the execution of SPECjbb2005, we want to see how much of the memory is region allocated and what is the impact on overall execution time.

The SPECjbb2005 is organized as a collection of Warehouses processing transactions. The core of the benchmark is a TransactionManager dispatching user transactions within a warehouse. We modified the benchmark to run for a fixed number of iterations instead of a fixed period of time and use the same random seed to ensure reproducibility. The starting number of warehouses is 4, the warehouse increment is 1 and the ending number of warehouses is 8. Configuring the benchmark this way causes the sequence of simulated warehouses to progress from the starting number to the ending number, incrementing by the increment value. We collected the results while running each warehouse for 100,000 iterations. The results do not change significantly when varying the number of iterations between 5,000 and one million.

The effort required to port code to our system is minimal. For the 12,581 lines of SPECjbb2005 code we only inserted 7 annotations. The annotations were inserted to map the execution of each transaction type to a region. Overall this results in around 3 millions region entries/exits for the chosen benchmark. The depth of runtime region stack reaches a maximum of 3.

9.1 Experimental Setup

We implemented the region analysis as a phase in the Substrate VM compilation pipeline. The results generated by the region analysis, allocation-site-to-region mappings and region activation/deactivation points, are used by the runtime system to drive the region allocation.

We run the benchmarks in three configurations:

- *Normal GC*: Generational GC without region allocation.
- *Hybrid allocation table*: Hybrid memory allocation with a region mapping allocation table for each allocation site.

- *Hybrid normalized*: Hybrid memory allocation with offset normalization, i.e., a unique region allocation offset for each allocation site.

We measure execution time, divided in collector and mutator execution time, allocated and collected memory, and number of partial and full garbage collections. Specifically for the region allocation configuration we measure the region freed memory and the maximum size of the region stack size. To extract the results we implemented a profiling infrastructure with a low performance impact. The execution and garbage collections times are measured using *System.nanoTime()*. The amount of collected memory is obtaining by summing the memory recalled by each garbage collection. The maximum region stack size is calculated before each region deallocation. We run each benchmark configuration 10 times and average over the results.

9.2 Summarized Results

In this section we look at garbage collection and execution time metrics for the selected benchmarks. To set up the experiment we choose a reasonable heap configuration size such that in the *Normal GC* configuration the garbage collection takes between 10% and 15% of the total execution time. For *Jolden* benchmarks we fixed the young generation size at 16 MByte young generation size and the old generation size at 128 MByte. For *SPECjbb2005* we fixed the young generation size at 64 MByte and the old generation size at 512 MByte.

In the figures presenting garbage collection results we don't distinguish between partial and full GCs. The execution of the *JOlden* benchmarks does not result in full GCs and for the *SPECjbb2005* we break down the numbers by GC type in the following section.

Figure 9.1 presents the percent of garbage-collected memory, out of total allocated memory. In the *Normal GC* configuration the reclaimed memory is less than 100% because it does

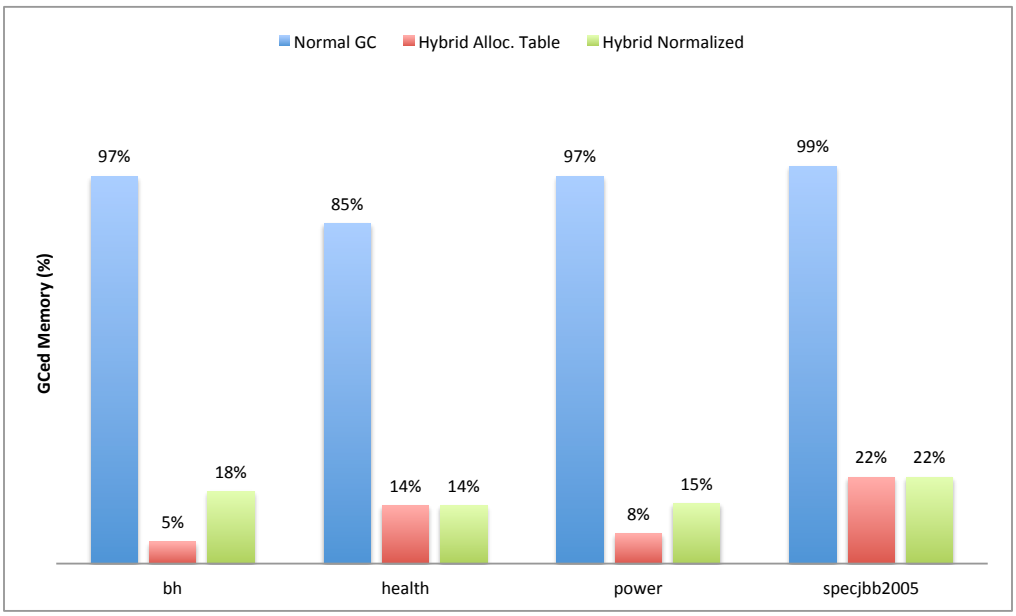

Figure 9.1: Garbage-collected memory.

not include the memory that is live when the application exits. As expected, the *Hybrid allocation table* configuration region frees a significant amount of the allocated memory. The garbage-collected memory is only 5% for the *jolden:bh* benchmark. The *Hybrid normalized* configuration still region frees most of the allocated memory, but, in general, less than the *Hybrid allocation table* configuration. This is due to normalization which effects in allocating conservatively in an older region than the optimal one. However, as it can be seen in Figure 9.4 normalization has a positive impact on overall execution time by optimizing the allocation fast path.

As expected, the reduction in garbage-collected memory results in less garbage collections. The results are shown in Figure 9.2. The total time spent doing garbage collection also decreases accordingly, as presented in Figure 9.3. The garbage collections speedup numbers are calculated by accumulating the times of all garbage collections.

Figure 9.4 presents the overall speedup. Although the garbage collection time is reduced significantly this does not result in overall speedups of the same magnitude. First, as stated earlier, the experiments are configured such that the GC time is less than 15% to begin with.

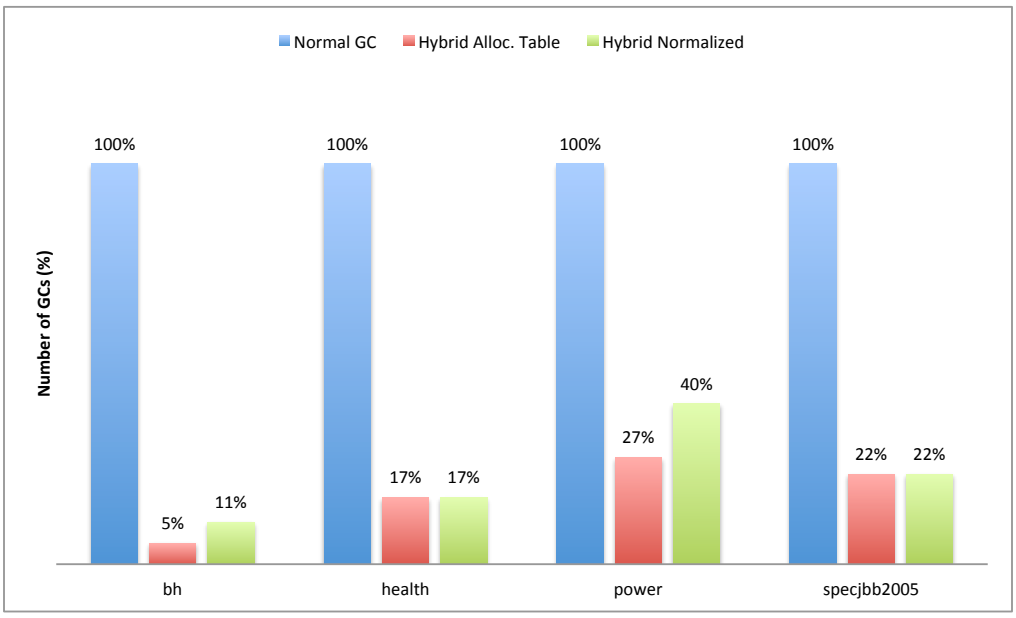

Figure 9.2: Number of garbage collections.

Thus, although the *Hybrid normalized* configuration execution results in speedups, these are capped at 15%. Second, the *Hybrid allocation table* uses an expensive table lookup on the allocation fast path. As expected this results in a overall execution slow down, which is as much as 7% for *SPECjbb2005*.

Analyzing the results we can conclude that the hybrid memory allocation scheme has potential in reducing GC pressure and improving the overall performance. Optimal region analysis results require expensive runtime table lookups which have a non-negligible impact on overall performance. Normalizing the allocation offsets results in some objects being allocated in non-optimal regions, however, it yields overall execution time savings.

The results suggest that an adaptive approach would be better suited. To preserve the benefits of optimal object-to-region mappings while not impacting the performance of fast path allocation we could use method cloning (discussed in Section 7.4) for the most frequent executed allocation sites. At the same time normalization could be used only for allocation sites that have a small enough difference between the smallest and the biggest allocation offset.

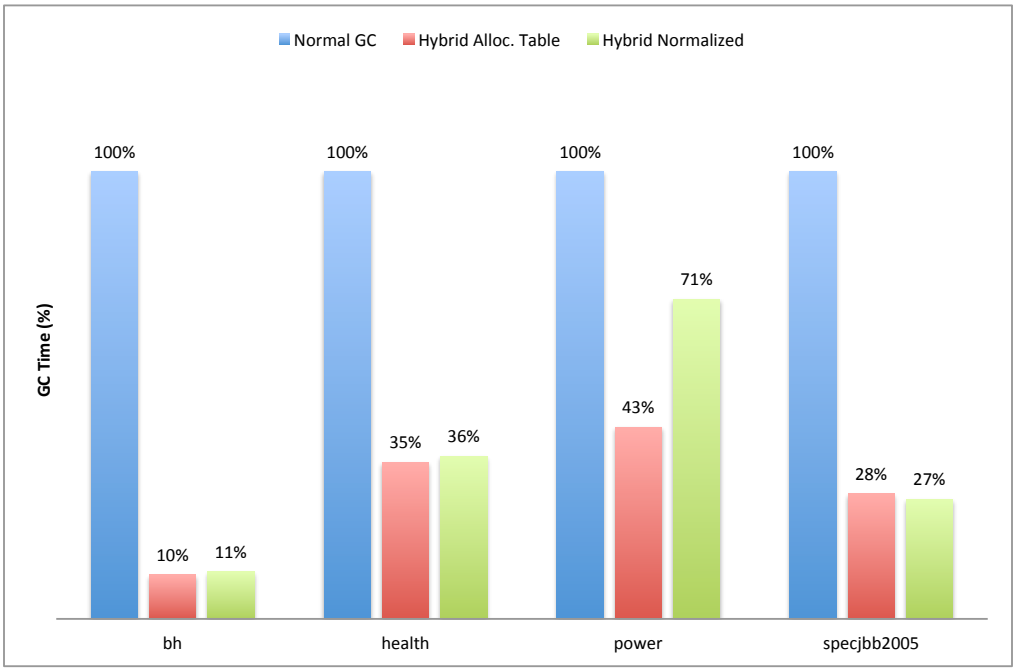

Figure 9.3: Garbage collections time.

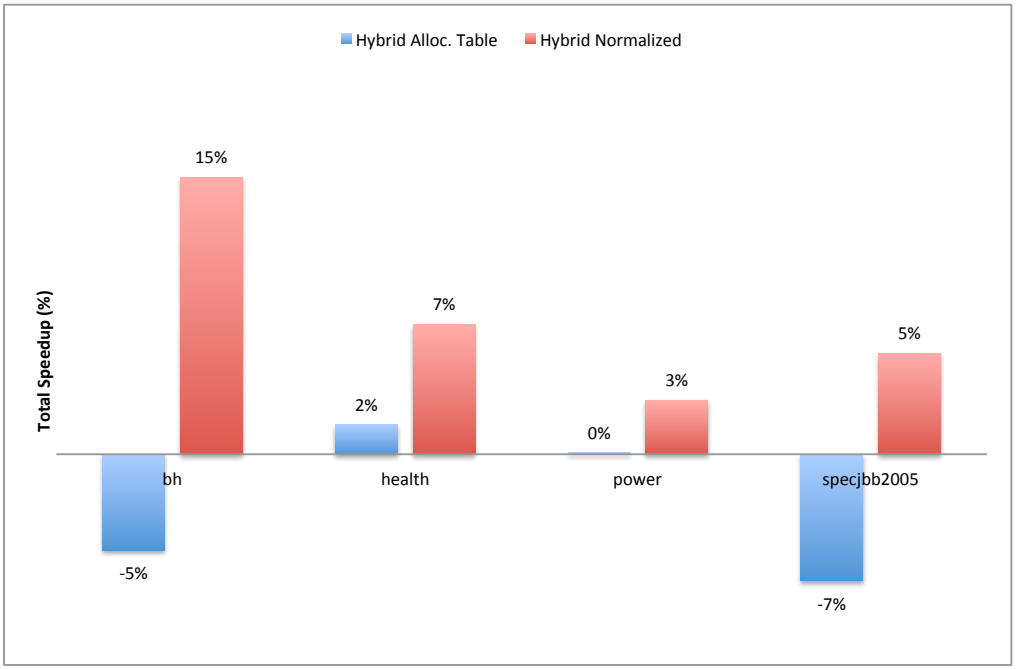

Figure 9.4: Execution time speedup.

9.3 Varying Young Generation Size

SPECjbb2005 is known to allocate a lot of memory during transaction processing, but only has a small set of long-lived objects (the data warehouses). Published benchmark results on the SPEC homepage typically use a multi-GByte young generation size to avoid full GC at all costs. We want to show that region-based memory management can significantly reduce the pressure on the incremental GC, i.e., we can achieve the same performance with a significantly smaller young generation size. Therefore we vary the young generation size only. Varying the old generation size would not add additional insights.

We fix the the old generation size to 512 MByte, and vary the young generation size from 1 MByte to 256 MByte. Each section of Table 9.1 is divided among the three configurations or only the hybrid configurations for region specific metrics. The columns represent the results for the various young generation sizes. Each benchmark configuration is run 10 times and the results are averaged.

The region metrics for the two hybrid configurations differ only slightly. The reason is that the region tree is balanced: there is a parent region mapped to the transaction dispatcher and child regions mapped to each type of transaction. Thus, the normalization performed for the hybrid normalized configuration preserves the optimal allocation region for all of the frequently executed allocation sites of SPECjbb2005.

The first rows of Table 9.1 show the allocation behavior. In our configuration, SPECjbb2005 allocates about 56 GByte of memory. This number scales mostly linearly with the number of executed transactions. We do not count memory allocations that are eliminated by escape analysis.

In the normal configuration, all allocated memory (apart from the heap that is present at application exit, roughly 0.5 GByte) is freed by the GC. In our hybrid configurations the GC

		Young generation size (MByte)								
		1	2	4	8	16	32	64	128	256
Allocated memory (MByte)	all configurations	56777	56777	56777	56777	56777	56777	56777	56777	56777
GC freed memory (MByte)	normal GC	56301	56396	56467	56520	56309	56375	56361	56315	56497
	hybrid alloc. table	12432	12373	12366	12174	12152	12279	12098	12237	12124
	hybrid normalized	12432	12373	12366	12174	12166	12276	12098	12237	12124
Region freed memory (MByte)	hybrid alloc. table	43446	43537	43581	43604	43615	43621	43623	43625	43625
	hybrid normalized	43446	43537	43581	43604	43615	43620	43623	43625	43625
Region freed memory (%)	hybrid alloc. table	77.30%	77.46%	77.54%	77.58%	77.60%	77.61%	77.61%	77.62%	77.62%
	hybrid normalized	77.30%	77.46%	77.54%	77.58%	77.60%	77.61%	77.61%	77.62%	77.62%
Max region stack size (KByte)	hybrid alloc. table	26	28	33	43	62	100	170	303	584
	hybrid normalized	26	28	33	43	61	99	170	303	583
Region enter #	hybrid	3000030	3000030	3000030	3000030	3000030	3000030	3000030	3000030	3000030
Region deallocation #	hybrid alloc. table	2987623	2993844	2996931	2998469	2999235	2999616	2999808	2999904	2999952
	hybrid normalized	2987623	2993844	2996931	2998469	2999235	2999616	2999808	2999904	2999952
Fulll GC #	normal GC	11	11	11	11	10	9	6	3	2
	hybrid alloc. table	10	10	10	8	5	3	1	1	0
	hybrid normalized	10	10	10	8	5	3	1	1	0
Incremental GC #	normal GC	28509	19002	11397	6326	3344	1719	871	439	219
	hybrid alloc. table	12572	6280	3139	1569	783	391	196	97	49
	hybrid normalized	12572	6280	3139	1569	784	391	196	97	49
Full GC time (s)	normal GC	4.08	4.12	4.20	4.29	3.98	3.44	2.52	1.28	0.95
	hybrid alloc. table	3.97	3.98	3.92	3.22	2.07	1.29	0.42	0.49	0.00
	hybrid normalized	3.91	3.91	3.96	3.17	2.06	1.28	0.42	0.50	0.00
Incremental GC time (s)	normal GC	92.44	68.06	46.76	31.54	22.88	15.82	10.34	5.93	4.68
	hybrid alloc. table	60.15	36.13	23.91	15.93	9.46	5.38	3.19	2.05	1.70
	hybrid normalized	47.62	31.00	21.21	14.52	8.80	5.06	3.02	2.07	1.64
Execution time (s)	normal GC	214.50	186.39	163.70	147.56	137.95	132.07	126.59	120.75	127.43
	hybrid alloc. table	197.98	171.59	160.97	151.85	143.48	138.17	135.80	133.38	135.92
	hybrid normalized	168.29	150.54	140.82	132.94	127.16	122.64	120.08	119.12	124.12
Speedup (%)	hybrid alloc. table	8%	8%	2%	-3%	-4%	-5%	-7%	-10%	-7%
	hybrid normalized	22%	19%	14%	10%	8%	7%	5%	1%	3%

Table 9.1: Memory allocation results for SPECjbb2005.

reclaimed memory is much smaller; a significant fraction moves to the region freed memory. For SPECjbb2005, about 78% of the memory is region freed.

This percentage varies slightly with the different young generation sizes. A small young generation leads to a higher number of incremental garbage collections (see later table rows). Since we treat the region memory as part of the young generation during GC, the GC frees a small amount of region-allocated memory. The fraction is small enough for it to not be a concern. The maximum size of region-allocated memory is never higher than 600 KByte. This number varies greatly with the young generation size. The frequent incremental col-

lections due to a small young generation size tend to empty the longer-living regions more often, limiting their maximum size.

9.3.1 Region Metrics

The number of entered regions shows how many times a region is pushed on the region stack. Since region placement only depends on the static analysis results, this number must be the same for all hybrid configurations and all heap sizes. The number of deallocations represents how many region exits perform memory deallocation, i.e., how many of the entered regions are exited before being scavenged by the garbage collection and invalidated (see Section 7.5). This number increases as the young generation size increases, since there are fewer garbage collections that can collect the region before it gets released. But even with small young generation sizes the vast majority of regions are exited before a garbage collection.

9.3.2 Execution Time

The second part of Table 9.1 focuses on GC count and execution time. The number of GCs and GC time decreases as the young generation size is increased. Our largest young generation size of 256 MByte has a total GC overhead of less than 2%, so presenting numbers for larger heap sizes would not add additional insights to the paper.

The hybrid memory allocation significantly decreases the number of incremental garbage collections, which also reduces the time spent performing GC. This is the expected benefit of hybrid memory management. The number of full garbage collections is also affected by the young generation size and the hybrid strategy. Every incremental GC promotes some short-living objects that happen to be alive at the time of GC to the old generation (we do not have multiple generations or aging of objects within the young generation). This fills up

the old generation, so eventually a full GC is needed. Hybrid memory management needs only about 1/4 of the young generation size to reach the same GC performance as with the normal GC. This matches our finding that about 3/4, i.e. 78%, of all memory is region managed.

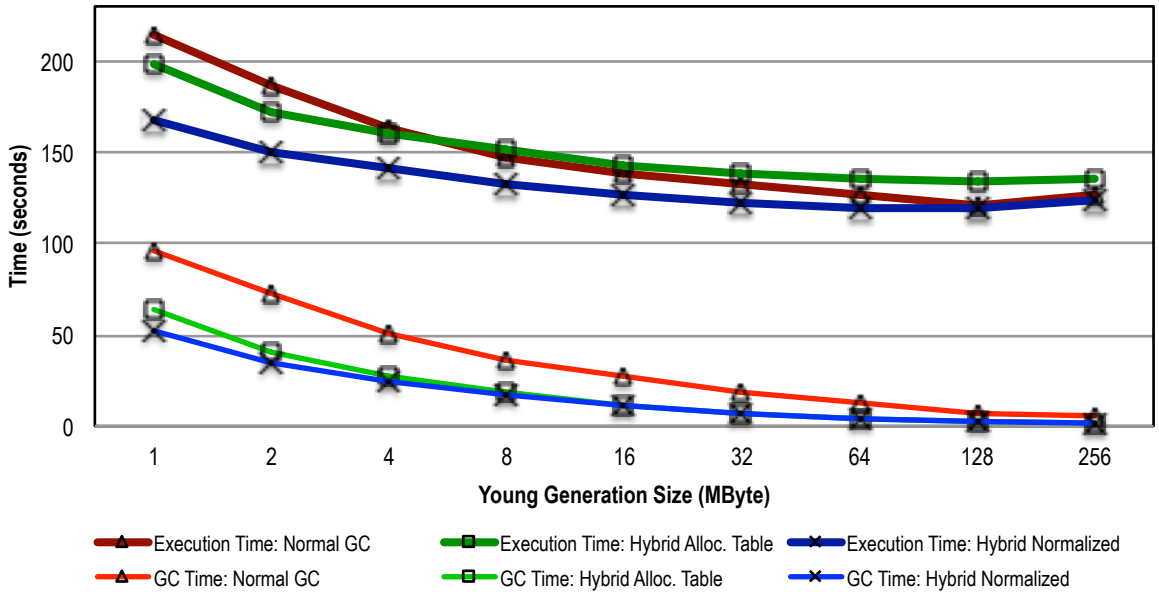

Figure 9.5: Execution and GC time for SPECjbb2005 with varying young generation size.

The improved GC time is also reflected in the overall execution time. The final lines of Table 9.1 show total execution time of the three configurations and the speedup of the hybrid memory configurations over normal GC. Figure 9.5 visualizes these numbers. The garbage collection time in Figure 9.5 includes both the full and incremental GCs time. The hybrid normalized configuration speeds up the execution time because it reduces the GC time. As expected, the *hybrid with allocation table* configuration is often slower than the other configurations: Since every allocation needs a table lookup to find the appropriate region, the mutator time is increased considerably. The hybrid normalized configuration does not add any overhead to allocation because the region offset is a compile time constant, i.e., no runtime computation is necessary to pick the allocation region.

9.3.3 Detailed Region Statistics

Table 9.2 shows the detailed region statistics for a young generation size of 256 MByte. It only contains the numbers for the *hybrid allocation table* configuration since the two hybrid configurations generate similar results. First two columns present the number of activations and actual deallocations. Next columns show the region total, region max, region average and region percent of total allocated memory. The region total represents the total memory freed by region exits. The region max represents the maximum memory allocated in the region at any time. The region freed percent is the fraction of the total region freed memory. Most of the memory is allocated in regions corresponding to customer transactions as intended. For the StartJBBthread region the number of activations is 30 but all other metrics are 0. This region is always garbage-collected before the region exit, so no region memory deallocation is possible. This is expected since the StartJBBthread is the topmost region in the region tree and the GC is triggered before this region ever gets the chance to exit and deallocate memory. Since no important allocation sites are in this region, switching to a GC scheme that preserves memory regions would not change the results significantly.

	Enter #	Deallocation #	Total (KByte)	Max (KByte)	Average (KByte)	Region freed %
StartJBBthread	30	0	0	0	0	0.00%
CustReportTxn	499521	499520	9569044	19	19	21.42%
DeliveryTxn	90913	90911	1665517	18	18	3.73%
NewOrderTxn	1318655	1318629	18337609	14	13	41.05%
OrderStatusTxn	90907	90906	1379782	15	15	3.09%
PaymentTxn	909093	909076	12498165	13	13	27.98%
RunStockLevelTxn	90911	90911	1222326	13	13	2.74%

Table 9.2: Detailed memory regions for SPECjbb2005 with a young generation size of 256 MByte.

Chapter 10

Related Work

10.1 Static Analysis

Our exploration of points-to static analysis is orthogonal to most of the previous work on this topic and complements previous results. For a comprehensive survey on points-to analysis and an in-depth discussion of the various flavors of context sensitivity we point the interested reader to the work of Smaragdakis and Balatsouras [47].

Prior work by Milanova et al. [34] has shown that an accurate points-to analysis for object-oriented languages such as Java must be context sensitive, i.e., a method must be analyzed separately for each calling context. A context insensitive analysis which considers a single copy for every method for all possible invocations is imprecise. The imprecision comes from object-oriented languages features and programming idioms such as encapsulation, inheritance and use of complex data structures (i.e., collections and maps).

The choice of context abstraction is also crucial for an accurate analysis. The trivial choice for context abstraction is invocation site, or chain of invocation sites. However, given that

instance methods work on encapsulated data using the implicit parameter `this` to read or write to instance fields a good context abstraction is an abstract representation of the receiver object [34]. A points-to static analysis that does not distinguish the different object structures accessed through the receiver object effectively merges the states of different objects and any access is reflected in all other objects across the same class. Empirical results by Smaragdakis et al. [49] show that in practice object-sensitivity has more impact on precision than call-site sensitivity. Since static methods do not have a receiver object, a hybrid approach where static method's context is based on chains of call sites was proposed and shows good precision at a low cost [27].

To further increase precision the depth of the calling contexts can be increased too. However, in practice a context sensitivity of 2 is the limit for an object-sensitive analysis [49] while a call site sensitive analysis of depth greater than 1 reaches the limits of practicability [34].

The quality of an object-sensitivity analysis is largely determined by its heap abstraction. Previous work by Liang et al. [32] has explored various static heap abstractions trying to find the design point that maximizes analysis precision. The base for all heap abstraction refinements is object allocation site to which additional bits of information are added.

Call stack heap abstraction adds the chain of the k most recent call sites on the stack of the thread creating the object. This is known as k-CFA with *heap cloning* [45]. Further refinements of *call stack* heap abstraction use the abstraction of the allocator method receiver object instead of the call site itself [34]. Another refinement of heap abstraction is *object recency*; it adds the recency index of an object to the allocation site to distinguish the last r objects created at an allocation site [4]. Heap abstraction based on *heap connectivity* tries to distinguish objects by their connectivity properties in the heap. It tries to associate objects with other objects reachable through the heap graph [41]. The impact of the various refinements of heap allocation sites on analysis precision is however dependent on the client

that uses the analysis results. There is no single abstraction that performs efficiently for all clients.

Type-sensitive points-to analysis [49] trades precision for analysis cost to obtain better scalability by using coarser approximations of objects as context. It makes an unconventional use of types as context: the context types are not dynamic types of objects involved in the analysis, but instead upper bounds on the dynamic types of their allocator objects.

Increased precision is not always possible due to cost restrictions or due to application complexity that can lead to analysis explosion. To address scalability issues of points-to analysis Milanova et al. [34] proposed a coarse grain parametrization technique: i.e., different context depth for call and heap sensitivity, or selecting a subset of reference variables that should be analyzed context sensitively (e.g., implicit parameter `this` and return variables). The authors also propose using different context depths to optimize frequently used patterns. For example allocation sites in container classes (e.g., the array of hash entries in HashTable) could be analyzed with increased context depth for more precise results to avoid *sharing* of objects stored in different containers. Our work proposes a more flexible, fine grained analysis sweet spot selection based on the runtime feedback.

Smaragdakis et al. [48] addressed the analysis cost issue by introducing a set-based preprocessing step that puts the program in a normal form optimized for points-to analysis. The preprocessing step computes constraints at the points-to set level that result in a reduced analysis space by removing local variables and instructions

Java programs are distributed in a Java bytecode format. However, optimizing the Java bytecode directly is difficult due to the fact that bytecode instructions operate directly on an operand stack, and thus have implicit uses and definitions of stack locations. A representation in which a statement refers explicitly to the variables it uses is better suited for static analysis and other optimizations. The Soot [57, 56] Java optimization framework

uses Jimple [58], a 3-address intermediate representation that has been designed to simplify analysis and transformation of Java bytecode. Soot also supports a static single assignment (SSA) form variation of Jimple. Our static analysis operates on the Graal internal representation [15], a graph-based, SSA form IR that models both control-flow and data-flow dependencies between nodes. The input to the analysis phase is a canonicalized IR. In the canonicalization process Graal discovers and simplifies easily optimizable code patterns, e.g., trivially statically bindable virtual calls and statically decidable type checks.

Doop [9] is a widely used static analysis framework for Java. It implements a range of algorithms, including context insensitive, call-site sensitive, and object-sensitive analyses. Doop expresses the various analyses declaratively using the Datalog language. Declarative logic programming allows for a natural expression of static analysis rules. However, our analysis step is designed as a phase in the compilation pipeline and since it operates on the Graal IR we decided to implement it completely in Java. An imperative programming language presents some challenges in dealing with type set operations. Our internal data structures are highly optimized and thread safe. Plus our implementation organizes abstract objects in unique type sets, thus redundant object state propagation is easily avoided.

Dynamic Features

At this stage of our project we do not support Java dynamic class loading and reflection. Our static analysis implementation operates under a closed world assumption: the code that is visible at compile time must be a superset of the executed code. Dynamic loading may load code that is not available for analysis before the program starts. However, there are various approaches that could be used to overcome these limitations and increase the range of supported language features.

Bodden et al. [6] have extended the scope of the static analysis to dynamically loaded classes by running the application before compilation and recording the dynamically loaded classes. The discovered classes are included in the analyzed code and the analysis operates under the assumption that no other classes can be loaded at runtime. Livshits et al. [33] proposed a static reflection resolution algorithm, which approximates the target of reflective calls as part of call graph construction, complemented by user provided specifications for reflective calls that rely on application input. Hirzel et al. [22, 21] presented an online version of Andersen's points-to analysis [2] that executes alongside the program, as an extension to the Jikes RVM [1], an open-source Java Research Virtual Machine. Thus the analysis has access to the new code as its loaded.

10.2 Feedback Directed Optimizations

We propose using runtime feedback to tune points-to analyses for increased application performance. Feedback directed optimizations are largely used in JIT compilers.

Runtime profiling gives a concrete view of the application execution patterns as it runs enabling the compiler to apply efficient optimizations. Hölzle et al. [24] described a dynamic optimization technique that feeds back type information from the runtime system to the compiler. Using the concrete types the compiler can efficiently devirtualize and inline dynamically dispatched calls.

Feedback directed optimizations cannot however guarantee that the assumptions based on the profiled execution hold for the entire life span of the application. Hölzle et al. [23] described an efficient dynamic deoptimization technique that can be used to roll aggressively optimized code back to interpreter execution. The deoptimization only affects the procedure that needs to be deoptimized; all other code runs at full speed.

Feedback directed optimizations have been extensively used in Oracle's HotSpot [38] and IBM's Jalapeño [3] JVMs. Our system exploits the runtime profiles collected by the HotSpot VM. The Java HotSpot VM improves performance through optimizing frequently executed application code. It collects runtime profiles to guide optimizations of the client (C1) and server (C2) compilers. The Client Compiler [28] is used by default for interactive desktop applications where low startup and pause times are more important than peak performance. The Server Compiler [39] is tuned for peak performance and it applies more aggressive optimizations. The two compilers share the same runtime environment and utilize the same profile recording system.

10.3 Region Based Memory Allocation

In this section we briefly summarize previous work in region-based memory management that is directly related to our approach. The body of work in region-based memory management is vast and this is by no means an exhaustive list. We point the interested reader to the retrospective paper of Tofte et al. [54].

Most of the work in region memory management for Java addresses the particular case of real-time Java. In real-time Java, unlike the standard Java, garbage collection is rarely used due to the unpredictability of temporal behavior of dynamic memory collection which affects the real-time scheduling policies. The Real-Time Specification for Java (RTSJ) [7] proposes extensions to the syntax and semantics of Java attempting to make the execution more predictable. RTSJ introduces region memory allocation through scoped memory regions to eliminate the unbounded pauses caused by interference from the garbage collector. Regions are used explicitly through programming language directives.

The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access references to objects allocated in the garbage-collected heap. Boyapati et al. [8] use a static type system to guarantee that the runtime checks will never fail for well-typed programs. The safety guarantee makes it possible to remove the runtime checks and their associated overhead.

To address the problem of dynamic memory management in real-time Java, [42] propose a static region inference algorithm coupled with region-based memory management. They involve the developer in the analysis process by providing feedback on programming constructs likely to produce memory leaks.

Nguyen et. al. [36] introduce the FACADE compiler framework for Java that automatically generates highly-efficient data manipulation code. They target Big Data systems where there is a clear distinction between an application's control path and data path. The control path organizes tasks into pipelines and performs optimizations while the data path represents and manipulates data. Their technique reduces the memory management cost by statically bounding the number of heap objects representing data items. Data is stored in the off-heap, native, unbounded memory while heap objects are created as facades only for control purposes. This leads to reduced memory management cost and improved scalability.

Qian and Hendren [40] introduce the idea of an adaptive, region-based allocator for Java. They use a dynamic approach which does not require static analysis or programmer annotations. They start by assuming that the scope of each method is mapped to a memory region and that all allocated objects are local to their allocator method region. Using runtime write barriers they catch escaping objects and adapt accordingly by marking the allocation site as non-local for future allocations.

Cherem and Rugina [12] propose a region analysis and transformation system for Java. First the analysis determines fine grained memory regions then the compiler transforms the

input program into an equivalent program with region-based memory management. Their approach allows dangling references and uses non lexically scoped regions.

In their masters thesis Christiansen and Velschow [13] explored region allocation for a subset of Java using explicit region annotations. The subset of Java leaves out features like concurrency, arrays and exception handling. They rely on language constructs for allocating, updating and deallocating regions.

Gay and Aiken [17, 18] explore extending C with explicit region annotations. They prevent unsafe region deletions by keeping a count of references to each region. By making the structure of a program's regions more explicit using type annotations, they reduce the overhead of reference counting. There is also the work on region-based memory management in Cyclone [19], a type safe dialect of C. Cyclone uses a combination of explicit annotations, implicit default annotations and local type inference.

Chapter 11

Conclusions

We explored the design and implementation of a hybrid, garbage-collected and region-based, automatic memory management for Java. This approach can be used to reduce the garbage collection pressure for transaction oriented programs. The hybrid memory management scheme is made possible by our formulation of a memory region aware points-to analysis that maps objects to memory regions. We implemented the analysis into the compilation pipeline of an existing virtual machine and modified its runtime system to enable region allocation.

We started our study by exploring various points-to analysis refinements aimed at improving results accuracy. Focusing on context sensitive points-to analysis formulations we compared points-to analysis results with runtime profiling to further validate the precision of the analysis results. We shifted the base of comparison from context insensitive analysis to runtime profiling to better understand what is the optimization potential of points-to analysis when targeting performance.

We formulated a memory region aware points-to analysis based on user inserted region annotations dividing the application execution into phases. The region analysis tries to

prove that objects do not escape the execution phase in which they are allocated. It extends a context sensitive points-to analysis and uses the region annotations as additional context elements. We presented analysis optimizations that ensure safe and efficient object allocation and garbage collection. The region analysis results can be summarized into a single, compile-time constant integer and that can be used to reduce the overhead of region allocation. Potential dangling pointers are prevented by constraining the analysis to respect a memory safety invariant. The invariant ensures that when a region is deallocated there are no pointers into it. It does this at the analysis time to avoid any unnecessary runtime overhead incurred by a dynamic check. We discussed garbage collection modifications necessary to avoid regions size explosion and ensure region allocation safety. Our region analysis formulation requires that regions can grow flexibly to accommodate allocation of objects. When the size of the region-allocated memory grows beyond a threshold an incremental collection must be triggered to scavenge the regions.

To validate our technique we implemented it in Substrate VM, an ahead-of-time compilation system and virtual machine for Java, but the results are not limited to this context. Substrate VM uses a static analysis step to discover the reachable world (classes, methods and fields) ahead of time. The analysis operates on the control flow graphs generated by the Graal compiler. We started by extending the existing analysis to a context sensitive points-to analysis. Then we implemented the region analysis as an independent phase that operates on the results generated by the core points-to analysis. We extended the generational garbage collector to support region allocation. Substrate VM's heap is organized in spaces composed of linked lists of chunks. Spaces can flexibly grow to accommodate for object allocation, thus were reused to implement regions. The region-allocated memory is conceptually part of the young generation and is scavenged together with it when the allocation threshold is reached.

In this thesis we presented compelling evidence that using region allocation can result in significant reduction in memory management costs. Our experiments show that the amount

of garbage-collected memory can be reduced by as much as 78%. Moreover, our results prove that region-based memory management can significantly reduce the pressure on the incremental GC, i.e., we can achieve the same performance with a significantly smaller young generation size.

11.1 Acknowledgments

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014.

Bibliography

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 47–65. ACM, 2000.

[4] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *Proceedings of the International Static Analysis Symposium*, pages 221–239. Springer-Verlag, 2006.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190. ACM Press, 2006.

[6] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the International Conference on Software Engineering*, pages 241–250. ACM Press, 2011.

[7] G. Bollella. *The Real-time Specification for Java*. Addison-Wesley Java Series. Addison-Wesley, 2000.

[8] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 324–337. ACM Press, 2003.

[9] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 243–262. ACM Press, 2009.

[10] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291. IEEE Computer Society, 2001.

[11] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 29–38. ACM Press, 1995.

[12] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the ACM International Symposium on Memory Management*, pages 85–96. ACM Press, 2004.

[13] M. V. Christiansen and P. Velschow. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, May 1998.

[14] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing Static Single Assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[15] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[16] Ecma TC39. ECMAScript test262.

[17] D. Gay and A. Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323. ACM Press, 1998.

[18] D. Gay and A. Aiken. Language support for regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80. ACM Press, 2001.

[19] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293. ACM Press, 2002.

[20] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152. ACM Press, 2002.

[21] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems*, 29(2), 2007.

[22] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 96–122. Springer-Verlag, 2004.

[23] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[24] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 326–336. ACM Press, 1994.

[25] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[26] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., 1996.

[27] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434. ACM Press, 2013.

[28] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization*, 5(1):7:1–7:32, 2008.

[29] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, 2006.

[30] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008.

[31] D. Liang, M. Pennings, and M. J. Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 6–12. ACM Press, 2005.

[32] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–427. ACM Press, 2010.

[33] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the Asian Conference on Programming Languages and Systems*, pages 139–160. Springer-Verlag, 2005.

[34] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[35] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319. ACM Press, 2006.

[36] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 675–690. ACM, 2015.

[37] OpenJDK. Graal Project.

[38] OpenJDK. HotSpot.

[39] M. Paleczny, C. Vick, and C. Click. The Java HotSpot server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology*. USENIX Association, 2001.

[40] F. Qian and L. Hendren. An adaptive, region-based allocator for Java. In *Proceedings of the ACM International Symposium on Memory Management*, pages 127–138. ACM Press, 2002.

[41] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 105–118. ACM Press, 1999.

[42] G. Salagnac, C. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time Java embedded systems. In *Embedded and Real-Time Computing Systems and Applications*, pages 73–80. IEEE Computer Society, 2007.

[43] M. Sharir and M. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, pages 189–234, 1981.

[44] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. Specjvm2008 performance characterization. In *Proceedings of the SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35. Springer-Verlag, 2009.

[45] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, 1988.

[46] O. G. Shivers. *Control-flow Analysis of Higher-order Languages of Taming Lambda*. PhD thesis, Carnegie Mellon University, 1991.

[47] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[48] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 253–270. ACM Press, 2013.

[49] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 17–30. ACM Press, 2011.

[50] SPEC. SPECjbb2005, 2005.

[51] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Comparing Points-to Static Analysis with Runtime Recorded Profiling Data. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 157–168. ACM Press, 2014.

[52] C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Safe and Efficient Hybrid Memory Management for Java. In *Proceedings of the ACM International Symposium on Memory Management*. ACM Press, 2015.

[53] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ACM International Symposium on Memory Management*, pages 18–24. ACM Press, 2000.

[54] M. Tofte, L. Birkedal, M. Elsman, and N. Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, Sept. 2004.

[55] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM, 1984.

[56] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java optimization framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 125–135. IBM Press, 1999.

[57] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34. Springer-Verlag, 2000.

[58] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations, 1998.