Chameleon: Adaptive Selection of Collections

Ohad Shacham
Tel Aviv University

Martin Vechev
IBM Research

Eran Yahav
IBM Research

Abstract

Languages such as Java and C#, as well as scripting languages like Python, and Ruby, make extensive use of Collection classes.

A collection implementation represents a fixed choice in the dimensions of operation time, space utilization, and synchronization. Using the collection in a manner not consistent with this fixed choice can cause significant performance degradation.

In this paper, we present CHAMELEON, a low-overhead automatic tool that assists the programmer in choosing the appropriate collection implementation for her application. During program execution, CHAMELEON computes elaborate trace and heap-based metrics on collection behavior. These metrics are consumed on-the-fly by a rules engine which outputs a list of suggested collection adaptation strategies. The tool can apply these corrective strategies automatically or present them to the programmer.

We have implemented CHAMELEON on top of a IBM's J9 production JVM, and evaluated it over a small set of benchmarks. We show that for some applications, using CHAMELEON leads to a significant improvement of the memory footprint of the application.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]; D.3.3 [*Language Constructs and Features*]

General Terms Performance, Languages

Keywords bloat, collections, java, semantic profiler

1. Introduction

Programming languages such as Java, C#, Python and Ruby include a collection framework as part of the language runtime. Collection frameworks provide the programmer with abstract data types for handling groups of data (e.g, Lists, Sets, Maps), and hide the details of the underlying data-structure implementation.

Modern programs written in these languages rely heavily on collections, and choosing the appropriate collection implementation (and parameters) for every usage point in a program may be critical to its performance.

Real-world applications may be allocating collections in thousands of program locations, making any attempt to manually select and tune collection implementations into a time-consuming and often infeasible task. It is therefore not surprising that recent studies [22] have shown that in some production systems, the utilization of collections might be as low as 10%, that is, 90% of the space consumed by collections in the program is overhead.

PLDI'09, June 15–20, 2009, Dublin, Ireland.
Copyright © 2009 ACM 978-1-60558-392-1/09/06...$5.00.

Existing profilers ignore collection semantics and memory layout, and aggregate information based on types. Offline approaches using heap-snapshots (e.g., [21, 22]) lack information about access patterns, and cannot correlate heap information back to the relevant program site.

In this paper, we present the first practical tool that automatically selects the appropriate collection implementations for a given application. Our tool uses what we call *semantic profiling* together with a set of collection selection rules to make an informed choice. This approach is markedly different from existing profiling tools where the user is forced to manually filter massive amounts of irrelevant data, typically offline, in order to make an educated guess.

Semantic Collections Profiling The semantic profiler consists of a tightly integrated collections-aware production virtual machine and a runtime library. During program execution, these two components collect a myriad of complementary *context-specific* collection-usage statistics such as *continuous* space utilization and access patterns for each object. This information is obtained online and transparently to the programmer, without any need for an offline analysis of a general (non-targeted) heap dump. The ability of CHAMELEON to map all statistics back to the particular allocation context in the program is extremely useful as it enables the developer to focus on collections with maximum benefit. We have also pre-equipped our tool with a set of collection selection rules which are evaluated on the dynamic statistics. The output of the tool is a set of suggestions on how to improve the collections allocated at a particular allocation context. We are not aware of any other tool that can automatically produce such effective information in a low-overhead manner.

Selection from Multiple Implementations In this work, we assume that we are given a set of interchangeable implementations for every collection type. The requirement is that the different implementations have the same logical behavior. For example, a Set may be implemented using an underlying array, or a linked-list, but all implementations have to maintain the functional behavior of a set (e.g., have no duplicates). We focus on optimizing the choice of collection implementation, and do not address the orthogonal problem of showing that two collection implementations are logically equivalent (as done, e.g., in [20]).

Our library provides a number of alternative implementations, and we allow the user to add her own implementations, and implementations obtained from other sources (e.g., [1, 2, 3, 4]).

1.1 Main Contributions

The main contributions of this paper can be summarized as follows:

- A *semantic profiler* which tracks useful collection usage patterns across space and time. The profiler aggregates and sorts data for each collection allocation-context.
- A collection-aware garbage collector which continuously gathers statistics for a collection ADT rather than individual objects.

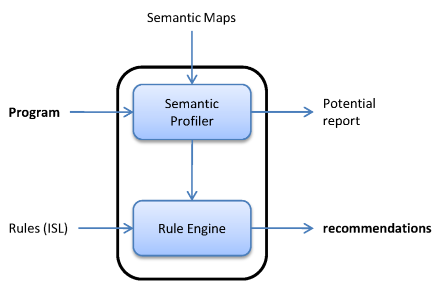

Figure 1. Tool overview

This is very useful as collection ADTs usually consists of several objects (that can be described by maps). The collector is *parametric* on the semantic ADT maps, and can be reused for any (including user-specific) collection implementation.

- A flexible rule engine that selects the appropriate collection implementation based on the profiling information. Our *rule engine* allows the programmer to write implementation selection rules over the collected profile information using a simple, but expressive implementation selection language.
- A complete implementation of our tool over a production JVM.
- Evaluation of our tool on a small set of benchmarks where we show that following CHAMELEON recommendations can lead to significant improvement in program space requirements, as well as running time.

2. Overview

In this section we provide a high-level overview of CHAMELEON by demonstrating how it is applied to an example, and briefly discuss some of the tradeoffs of collection implementations.

2.1 Motivating Example

TVLA [18] is a flexible static analysis framework from Tel-Aviv University. The framework performs abstract interpretation with parametric abstractions, and computes a set of abstract states that over-approximate the set of all possible concrete program states. TVLA is a memory-intensive application, and its ability to tackle key verification challenges such as concurrent algorithms (which have large state spaces) is mostly limited by memory consumption. The TVLA framework makes extensive use of collections.

Our goal in this example is to optimize the collections usage in TVLA. The first step towards that goal is to check the potential for collection optimizations in the application.

Fig. 1 shows an overview of CHAMELEON. The tool works in two automated phases: (i) semantic collection profiling—gathering a wide range of collection statistics during a program run; (ii) automatic selection using a rule engine—using a set of selection rules evaluated over the collected statistics to make implementation selection decisions. The tool is parametric on the semantic maps used for profiling (Sec. 3.2), and on the set of selection rules (Sec. 3.3).

Fig. 2 shows the percentage of live-data that is consumed by collections in TVLA running on a particular analysis problem—-showing that Lindstrom's binary-search-tree traversal preserves the shape of the underlying tree. This figure is the actual output as produced by the semantic profiler in CHAMELEON.

The figure shows three measures as percentage of the live data: (i) total live data consumed by collections ($live$); (ii) total size of the used parts of collections ($used$); (iii) lower-bound space consumption of the actual collection content ($core$).

The gap between $live$ and $core$ is the best-case potential space saving for collections (see more details on the nature of this space

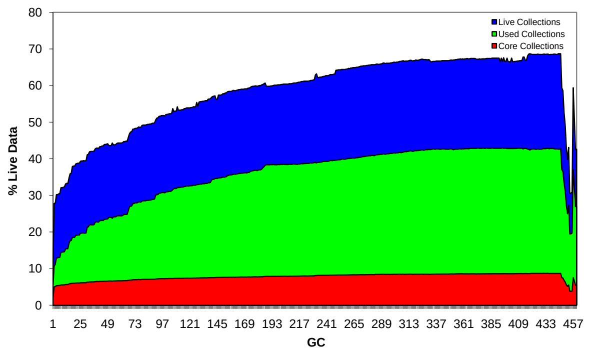

Figure 2. Percentage of live data consumed by collections in TVLA.

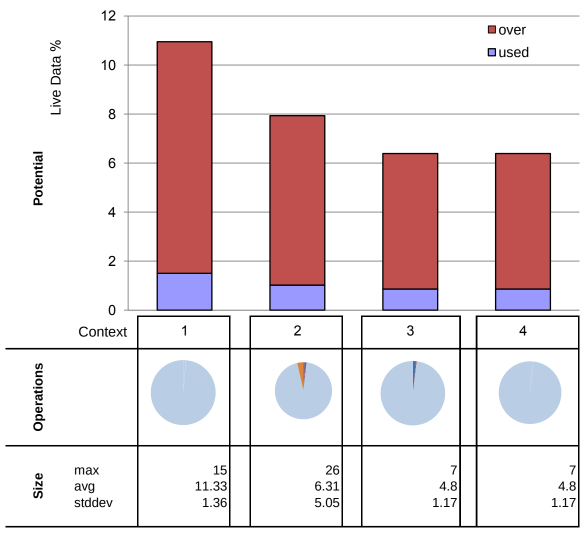

Figure 3. Combined results for top 4 allocation contexts in TVLA.

overhead in Section 2.2). Of course, some collection implementations, such as hash tables, introduce additional space in order to facilitate efficient operations, so comparison to $core$ is non realistic. We provide the $core$ measure as a lower-bound for the space requirement, and to see how changes to the volume of the actual stored data affect the space consumed by collections.

The gap between $live$ and $used$ corresponds to the total space allocated by the collection implementation that is not used to store application entries. In the figure, we see that collections constitute up to 70% of the live data, and the part used to store collection elements is only up to 40% of the live data.

At this stage, it seems that there is a realistic potential for space saving, but the question is — how do we realize this potential? In particular, how do we relate this information to the program? What can be done in order to avoid this space overhead? What points in the program should be modified?

Using several heap-snapshots taken during program execution may reveal the types that are responsible for most of the space consumption. However, a heap snapshot does not correlate the heap objects to the point in the program in which they are allocated. Therefore, finding the program points that need to be modified requires significant effort, even for programmers familiar with the code. Moreover, once the point of collection allocation is found, it is not clear how to choose an alternative collection implementation. In particular, choosing an alternative collection implementation with lower space overhead is not always desirable. Some structures, such as hash-tables, have inherent space overhead to facilitate more time-efficient operations. In order to pick an appropriate implementation, some information about the *usage pattern* of the collection in the particular application is required.

CHAMELEON is the first tool to integrate heap-information with information about the usage-pattern of collections. The semantic profiler in CHAMELEON produces a ranked list of allocation contexts in which there is a potential for space saving. For each such allocation context, the profiler CHAMELEON provides comprehensive information such as the distribution of operations performed on collections allocated at the context, the distribution of collection sizes, etc. Fig. 3 shows an example for such a summary. It shows the top 4 allocation contexts in TVLA, with their corresponding space saving potential. For example, for context 1, there is a space potential of roughly 10 percent of total live heap. Additionally, for each context, the tool provides the distribution of operations (represented as circles in the figure). For brevity, we don't show the names of the operations. For contexts 1, 3 and 4, the operation distribution is entirely dominated by `get` operations, while for context 2 there is also a small portion of `add` and `remove` operations. In addition to profiling information for each context, CHAMELEON produces suggestions on which collection implementations to use. For this example, we get the following succinct messages (for brevity we only show suggestions for contexts 1 and 4):

- **1:** HashMap:tvla.util.HashMapFactory:31;tvla.core.base.BaseTVS:50
  - replace with ArrayMap
- **4:** ArrayList:BaseHashTVSSet:112; tvla.core.base.BaseHashTVSSet:60
  - set initial capacity

To produce this report, CHAMELEON combines information on how the collections are used, with information on the potential saving in each context. The combined information is used by CHAMELEON's rule-engine, to yield collection tuning decisions that are presented to the user. The final report usually includes the precisely tracked context, which in our case consists of the call-stack when allocation occurred (usually of depth 2 or 3). This is often required when the application uses factories for creation of collections (as is done sometimes in TVLA).

Next, we apply the collection decisions CHAMELEON advocated in 5 top allocation sites in TVLA, and re-run the application. The overall effect on the total space required by the application is dramatic. In particular, the minimal heap-size required to run the application has been reduced by 50%. The effect on the overall running time is also significant. The total time required to complete the verification by using the modified version based on the collection implementations advocated by CHAMELEON is more than 2.5 times faster than the original one (from 49 to 19 minutes). Further saving of time and space is possible by modifying additional program points.

In contrast to standard profiling tools which require heavy manual involvement, by using the semantically focused, context-specific suggestions provided by CHAMELEON, we were able to achieve dramatic performance improvements quickly and with little manual effort.

2.2 Tradeoffs in Collection Implementations

Selecting an appropriate collection implementation is more complicated than it seems at first sight.

Time It is possible to base the selection on asymptotic time complexity of collection operations. However, the asymptotic time complexity of collection operations is not a good measure of their behavior when the collections contain a small number of items. In the realm of small sizes, constants matter. Furthermore, in practice, the actual performance of a collection is affected by different aspects, such as the locality of the selected structure, the cost of computing a hash function, cost of resizing the structure etc.

Space Collections vary in how much space overhead is consumed for storing a specific amount of data. They typically have different fixed overhead per element in the collection. For example, every element stored in the `LinkedList` implementation has an `Entry` object associated with it, where the Entry object stores a reference to the actual element, and two references to the next and previous entries in the list.

At each allocation site in the program, we define the utilization of a data structure as the ratio between the size of the data that it represents and the total amount of memory that this instance currently uses. Similar utilization metrics are used in the context of memory health measures [22]. As utilization varies during the execution, we consider both the utilization along points of program execution, and the overall average utilization of the collection.

There are several causes of low utilization: (i) the initial capacity of the collection is not suited to the average size of data stored in it; (ii) the collection is not compacted when elements are removed from it; (iii) high overhead per item in the collection.

For example, an `ArrayList` expands its capacity whenever it runs out of available space. The capacity grows by the function $newCapacity = (oldCapacity * 3)/2 + 1$. Consider an `ArrayList` that has an initial capacity of 100 and contains 100 elements. Adding another element increases the size of the allocated array to 151 while only containing 101 elements.

Space/Time Tradeoffs It is key to note the tradeoff between time and utilization (space). We can improve utilization by taking more time to perform operations. For example, given an `ArrayList` implementation, we can resize the array on every operation exactly to the number of elements it contains. This would incur a significant time penalty, but would keep the utilization at close to 100% (accounting for the meta-data in the collection object header etc.). Conversely, if we don't care about utilization, we can pre-allocate the array at the maximal number of elements, which would yield a very low utilization, but would avoid the need for resizing the array. Similarly, choosing an array over a linked-list would improve utilization, but would make update operations more costly.

2.3 Possible Solutions for Low Utilization

There are several seemingly reasonable solutions that can be used to tackle the poor utilization of data structures.

First, we can set the initial size of all allocated collections to one and then resize the collection size whenever an insertion or removal operation takes place. Second, we can use a hybrid collection mechanism. Initially the structure is implemented as an array. Then, whenever, the size of the collection increases beyond a certain bound, we can convert the array structure to the original implementation.

The advantage of both of these solutions is that they operate based only on local knowledge. That is, decisions for the collections implementation and size are determined within the specific collection object and are not based on any kind of global information such as allocation context.

Unfortunately, we were unable to reduce the memory footprint using these solutions with a reasonable time penalty.

Using small initial sizes does not reduce the memory footprint due to the fact that in Hash-based ADT, such as HashMap, each hash entry is represented by a new object containing three pointer fields. The first is a next pointer referencing the next entry. The second is a prev pointer referencing the previous entry. The third is a pointer to the data itself. The entry object alone on a 32-bit architecture consumes 24 bytes (object header and three pointers). Therefore, even when starting with a small initial size, significant memory not related to actual data is consumed, in this case, due to the large entry size.

The second (hybrid) solution can be effective in reducing footprint; however, choosing the size when the conversion from an array based implementation should take place is very tricky without causing significant runtime degradation. In TVLA for example, making the conversion of ArrayMap to HashMap at size 16 provides a relatively low footprint with 8% performance degradation. However, increasing the conversion size to a larger number than 16 does not provide a smaller footprint and leads to performance degradation. Moreover, reducing the conversion size to 13 provides the same footprint as the original implementation does.

3. Automated Collection Selection

In this section, we discuss our solution for automatically selecting the appropriate collections for a given user program. First, in Section 3.1, we define the problem. Then, we show how we address this problem with a combination of semantic collection profiling (Section 3.2), and a rule engine (Section 3.3).

3.1 Optimal Selection of Collection Implementations

Given a program that uses collections, our goal is to find an assignment of collection implementations that is *optimal for that program*. An optimal choice of collection implementations tries to balance two dimensions: minimizing the time required to perform operations while also minimizing the space required to represent application data.

The problem of optimal collection selection can be viewed as a search problem: for every point in a program allocating a collection, for each possible collection implementation, run the program, and compare the results in terms of space consumption and overall running time. However, this approach is not likely to scale for anything but the smallest programs. Furthermore, comparing results across executions is a daunting task in the presence of non-determinism and concurrency.

An alternative approach is to select collection implementations based on collection *usage statistics* extracted from the client program. Since there is no a priori bound on the number of collection objects in a program, and there is no a priori bound on the sequence of operations applied on a collection object, it is not practical to represent all operation sequences directly, and an abstraction of the usage patterns is required.

In principle, an abstraction of the collection usage pattern in a program can be obtained either statically or dynamically. However, static approaches to this problem typically abstract away the operation counts, which are a crucial component of usage patterns, and are not likely to scale to realistic applications. Seeking a scalable approach, we focus our attention on selection based on dynamic information. A dynamic approach would have to track, in a scalable manner, enough information on the usage of collections to enable the choice of appropriate implementations.

3.2 Semantic Collections Profiling

In this section we describe the information collected by the semantic collections profiler of CHAMELEON and how this information is used in order to make collection selection decisions.

3.2.1 Allocation Context

Our work is based on the hypothesis that the *usage patterns* of collection objects allocated at the same allocation context are similar. More precisely, we define the *allocation context* of an object o to be the allocation site in which o was allocated, and the call stack at the point when the allocation occurred.

Name	Description
Overall live data	Total/Max size of all reachable objects
Collection live data	Total/Max size of collection objects
Collection used data	Total/Max used part of collection objects
Collection core data	Total/Max core part of collection objects
Collection object number	Total/Max # of live collection objects
Number of operations	Total number of operations performed
Avg/Var operation count	Average # of times an operation was performed, and its standard deviation.
Avg/Var of maximal Size	Average maximal size of collections allocated, and its standard deviation.

Table 1. Heap and trace statistics gathered by CHAMELEON for each execution. Information is aggregated per *allocation context*.

For allocation contexts in which we observe similarity between usage patterns to hold within reasonable statistical confidence, we determine the type of collections that should be allocated in the context based on the average usage pattern.

DEFINITION 3.1 (Stability). *We define the stability of a metric in a partial allocation context c as the standard deviation of that metric in the usage profile of collections allocated in c.*

Examples of metrics are: the number of times a certain operation is performed on a collection instance and the maximal size of the collection during its lifetime. For every metric we define a threshold that determines the limit under which the metric is considered stable.

Practically, the full allocation context is rarely needed, and maintaining it is often too expensive. Therefore, we use a *partial allocation context*, containing only a call stack of depth two or three. The observation that (a small) allocation context is crucial to object behavior is inline with the recent study of Jones et. al [15]. In that study, the authors observed that objects allocated in the same context tend to behave similarly and a garbage collection strategy should be made aware of this correlation. In our work, we exploit this insight for optimizing collection usage.

3.2.2 Collection Statistics

CHAMELEON records a wide range of statistics indicating how collections in the program are used. Much of the information recorded by the tool is per allocation context, and is an aggregation of the information collected for objects allocated at that context.

Dynamically Tracked Data The tracked data is shown in Table 1. The collected information is a combination of information about the heap (e.g., the maximal heap size occupied by collection objects during execution), and information about the usage pattern (e.g., the total number of times `contains` was invoked on collections in the context).

Heap Information The heap information provides a comprehensive summary of the space behavior of collections during program execution. This information is collected on every garbage collection (GC) cycle. The GC computes the total and maximal live data of the program where the total live data is the sum of all live data's accumulated over all of the GC cycles and the maximal live data is the largest live data seen in any GC cycle. The GC has been augmented with *semantic maps* and routines to compute various *context-specific* collection information (discussed further in Section 4). First, it computes the total and maximal space consumed by reachable collection objects across all GC cycles. Second, it computes the total and maximal space actually used by these collection objects (collection used data). This is important for knowing how much of the collection object is really utilized. Thirdly, it computes the total and maximal collection core size, which would be the ideal

$$
\begin{array}{lll}
\mathit{rule} & := & \mathit{srcType}\ \mathit{cond}\ \mathit{implType} | \\
 & & \mathit{srcType}\ \mathit{cond}\ \mathit{implType}(\mathit{capacity}) \\
\mathit{srcType} & := & \mathit{Collection} \mid \mathit{ArrayList} \mid \mathit{LinkedList} \mid \ldots \\
\mathit{implType} & := & \mathit{ArrayList} \mid \mathit{ArrayMap} \mid \mathit{HashSet} \mid \ldots \\
\mathit{cond} & := & \mathit{comparison} \mid \mathit{cond} \wedge \mathit{cond} \mid \ldots \\
\mathit{comparison} & := & \mathit{expr} \leq \mathit{constant} \mid \mathit{expr} == \mathit{constant} \mid \ldots \\
\mathit{expr} & := & \mathit{opCount} \mid \mathit{opVar} \mid \mathit{data} \mid \mathit{expr} + \mathit{expr} \mid \ldots \\
\mathit{opCount} & := & \#\mathit{add} \mid \#\mathit{get}(\mathit{int}) \mid \#\mathit{get}(\mathit{Object}) \mid \ldots \\
\mathit{opVar} & := & @\mathit{add} \mid @\mathit{remove} \mid \ldots \\
\mathit{data} & := & \mathit{tracedata} \mid \mathit{heapdata} \\
\mathit{tracedata} & := & \mathit{size} \mid \mathit{maxSize} \mid \mathit{initialCapacity} \\
\mathit{heapdata} & := & \mathit{maxLive} \mid \mathit{totLive} \mid \mathit{maxUsed} \mid \mathit{totUsed} \ldots \\
\mathit{capacity} & := & \mathit{INT} \mid \mathit{maxSize}
\end{array}
$$

Figure 4. Simple language for implementation selection rules.

space that would be required to store the core elements of the collection object in an array. This statistic is useful to provide a lower bound on the space requirement for the content of the collection (hence indicating the limit of any optimization). Finally, the total and maximum number of live collection objects are computed.

Trace Information As mentioned earlier, recording the full sequence of operations applied to a collection object has a prohibitive cost. Instead, our trace information records the distribution of operations, as well as the maximal size observed for collections at the given context. The average operation counts provide a count of all possible collection operations. For brevity, we do not list all of them here. For some operations, those that involve interactions between collections, we introduce additional counters that count both sides of the interaction. For example, when adding the contents of one collection into another using the $c1.\mathit{addAll}(c2)$ operation, we record the fact that addAll was invoked on $c1$, but also the fact that $c2$ was used as an argument for addAll. Similarly, we record when a collection was used in a copy constructor. These counters are particulary important for identifying temporary collection objects that are never operated upon directly, other then copying their content.

Using Profiling Information The statistics from the tool can be used in several ways. For example, as the program runs, the user can request the tool to output the current top allocation contexts, sorted by maximum benefit. In the case where the user wants to make manual changes, she can focus on the most beneficial contexts instantly. Alternatively, she can use the recommendations automatically computed by the tool, which are based on a set of selection rules. To allow flexibility in querying the information collected by the tool, and select appropriate implementations based on it, we let the user write rules in a simple language. We describe those next.

3.3 Rule Engine

A Simple Rule Language We allow the user to write replacement rules, using the language of Fig. 4. The language is pretty standard, and in the figure we abbreviate rules that contain standard combinations of operations, such as boolean combinations for *cond* and arithmetic operators for expr. The language allows to write conditional expressions under which a replacement takes place. The conditional expressions use the metrics of Table 1 as the basic vocabulary. The language allows to write conditional expressions comparing the ratios between operation counts (e.g., the ratio of contains operations $\#\mathit{contains}/\#\mathit{allOps}$), the operation count itself (e.g., $\#\mathit{remove} == 0$) etc. It also allows the user to check the variance of counts (e.g, $@\mathit{add}$). The language also allows the user to query the live-data occupied by collections at the context, and the used-data occupied by collections at the context. These are typically used to figure out whether the potential saving in this allocation context ($\mathit{totLive} - \mathit{totUsed}$) is greater than some threshold.

3.3.1 Chameleon Collection Selection

Table 2 shows several examples of selection rules that are built into CHAMELEON. The constants used in the rules are not shown, as they may be tuned per specific environment. For example, the rule

$$\mathit{ArrayList}: \#\mathit{contains} > X \wedge \mathit{maxSize} > Y \rightarrow \mathit{LinkedHashSet}$$

specifies that if the type allocated at this context is an `ArrayList`, and the average number of `contains` operations performed on collections in this context is greater than some threshold X, and the average maximal size of the collection is greater than some threshold Y, then the selected type should be a `LinkedHashSet`. This rule corresponds to the fact that performing a large number of `contains` operations on large-sized collections is better handled when the collection is a `LinkedHashSet`. Of course, the rule can be refined to take other operations into account. The user can write various expressions in this language that dictate which implementation to select. For example, when the potential space saving is high, one may want to apply a different collection selection even if it results in a potential slowdown. For instance, the space benefit of the rule selecting an `ArraySet` instead of `HashSet` may outweigh the time slowdown when the potential space saving ($\mathit{totLive} - \mathit{totUsed}$) is greater than some threshold. Conversely, we can avoid any space-optimizing replacement when the potential space savings seems negligible.

Section 5 shows that using CHAMELEON recommendations based on rules such as those of Table 2 can yield significant performance improvements.

Stability If stability is not specified explicitly in the rule, it is assumed that any metric has to have its standard deviation less than a fixed constant (in the current implementation, size values are required to be tight, while operation counts are not restricted). Generally, different metrics may require different measures of variance based on their expected distribution. For example, while the operation counters usually distribute normally, maximal collection sizes are often biased around a single value (e.g., 1), with a long tail. Our current implementation uses standard-deviation as the stability measure, but in the future we plan to evaluate the suitability of other measures of variance for different metrics.

3.3.2 Towards Complete Automation

The current operation mode in which CHAMELEON is used is to evaluate all selection rules at the end of program execution, when complete information has been obtained for all collections allocated at a given context. The suggested implementations can then be applied by the programmer (or by the tool) and the program can be executed again (with or without profiling).

An interesting challenge is whether the act of replacement can be applied *while* the program is running. Such an online solution may be beneficial for several reason:

Lack of Stability: It is possible that collection objects from a given allocation context exhibit wide variation in behavior, for example due to different program inputs, phasing or non-determinism. Hence, detecting these cases and allocating the appropriate collection object may be more advantageous than sticking to a single implementation for all cases.

Optimization of Underlying Framework: Most real-world software makes use of framework code. The framework code itself may make extensive use of collection. Online selection can specialize the collection-usage in underlying frameworks, that is typically outside the scope of programmer's manual modifications. In gen-

Type	Condition	Category: Message	Suggested Fix
ArrayList	$\#contains > X \wedge maxSize > Y$	Time: Inefficient use of an ArrayList: large volume of contains operations on a large sized list	LinkedHashSet
LinkedList	$\#get(int) > X$	Time: Inefficient use of a LinkedList: large volume of random accesses using get(i)	ArrayList
LinkedList	$(\#add(int, Object) + \#addAll(int, Collection) + \#remove(int) + \#removeFirst) < X$	Space: LinkedList overhead not justified when adding/removing elements from the middle/head of the list is hardly performed	ArrayList
Collection	$maxSize == 0$	Space: Redundant collection allocation	LazyArrayList
HashSet	$maxSize < X$	Space: ArraySet more efficient than an HashSet. Time: operations on a small array might be faster than on an HashSet	ArraySet
Collection	$\#allOps = 0$	Space/Time: redundant collection	avoid allocation
Collection	$\#allOps == \#copied$	Space/Time: redundant copying of collections	eliminate temporaries
Collection	$maxSize > initialCapacity$	Space/Time: incremental resizing	set initial capacity
Iterator	$collection.size == 0$	Space: Redundant iterator	remove

Table 2. Example of built-in CHAMELEON rules.

eral, this follows a theme of specializing the library for a particular client, as part of the client's execution in the runtime environment.

No Programmer Effort: Manual replacement may require non-trivial code-modifications to deal with factories and deep allocation contexts. Dynamic selection is performed as part of the runtime environment and requires no manual modifications to the source code.

Dealing with completely automatic replacement is challenging because decisions may have to be based on partial information: at what point of the execution can we decide to select one collection implementation over another? For example, if the tool replaces the type allocated at a given context from a `HashMap` to an `ArrayMap` on the premise that objects allocated at that context have small maximal sizes, even a single collection with large size may considerably degrade program performance. Additionally, such a tool must run with sufficiently low overhead to be enabled during production deployment. Therefore, it is crucial to reduce overhead costs and in particular, it is vital to be able to obtain allocation context cheaply.

Towards the vision of fully automatic management of collections at runtime, we performed preliminary experiments where we used CHAMELEON in a mode where all replacements are done completely automatically at runtime, without any user involvement. We describe our results in Section 5.

4. Implementation

In this section we present the design and implementation of our tool. The tool consist of two complementary components: the library and the virtual machine, which are integrated in a manner that is transparent to the end user. The design of these components is such that they can be used separately by switching on and off each component on demand. However, for maximal benefit we typically use them together. By selectively instrumenting the library, we are able to record various useful statistics such as frequency of operations and distributions of operations for a given collection size. While this information is useful, it still does not provide us with a relative view of how collections behave with respect to the whole system. However, such global information can be extracted from the virtual machine and in particular from the garbage collector (GC). By instrumenting the GC to gather semantic information about collections, we are able to answer questions such as the total live data occupied by collections at a specific point in time. Such information, while cheap to obtain from the GC, is very costly to obtain at the library level. Next, we describe each component separately as well as how they interact with each other.

4.1 Design Choices

One of the core principles that we followed in our approach is to avoid as much as possible any changes to the original program. A key place where a dilemma between portability and slightly better efficiency occurs is during allocation of a collection object. For example, if the user program requests an allocation of a HashMap object and the system determines that for this context, it is best to implement that HashMap object with an ArrayMap, we are faced with two possible implementation choices. First, we can make ArrayMap a subtype of HashMap and then return ArrayMap. The problem is, that ArrayMap would then inherit all fields from HashMap. Further, any program expressions that depend on the precise type being HashMap may work incorrectly. Another solution is to have ArrayMap and HashMap as sibling types, but to return an object of type ArrayMap. In that case, we need to make sure that all type declarations in the program match ArrayMap (that were HashMap before) and that all semantic behavior depending on a specific type is preserved. This is the approach taken by Sutter et. al for details [25]. However, statically re-writing the type declarations of the program is intrusive, challenging, can lead to subtle errors due to language features such as dynamic typing, and is generally difficult to scale on large programs. Our solution in that case has been true to Lampson's statement that all problems in computer science can be solved by another level of indirection. Hence, we add another level of indirection between the program and the collection implementation. That is, each allocation of a collection object requires a wrapper. In our example, whenever HashMap is allocated, it will be a small wrapper object. Then, internally, the wrapper object can point to any implementation of HashMap. We believe that a small delta in inefficiency is worth the software reliability gains. Further, we believe that with VM support we can reduce this inefficiency further (e.g. via object inlining)

4.2 Library Architecture

Fig. 5 shows the architecture of the CHAMELEON libraries.

Our wrappers delegate collection operations to the underlying selected collection implementation (similar to the Forwarding types in Google's Collections [2]). The only information kept in the wrapper object is a reference to the particular implementation. In our solution, the actual backing implementation can be determined statically by the programmer (by explicitly providing the constructor with an appropriate constant), left as the default choice that the programmer indicated, or determined dynamically by the system.

As the wrapper allocates the backing implementation object, it also obtains the call stack (context) for this allocation site and constructs a VMContextKey object that records it (via the locationId fields inside it). This object is then used to look up the cor-

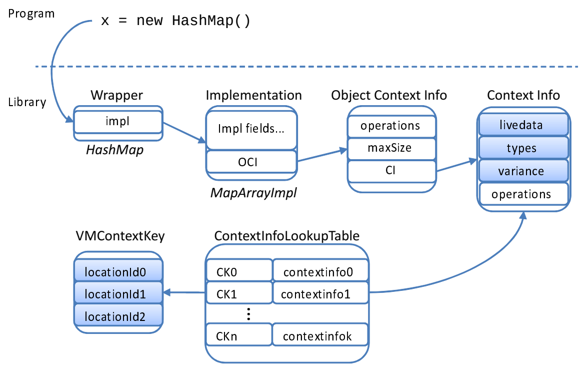

Figure 5. CHAMELEON library architecture. Shaded fields are updated by the VM.

responding `ContextInfo` object, which records aggregate information for this context. In order to collect information on the collection usage pattern for this context, the backing implementation may allocate an `ObjectContextInfo`. This object is used to store the various operation counters, collection maximal size, etc. When the collection implementation object dies, the contents of its object information object are aggregated into the corresponding `ContextInfo` object (via finalizers as described later).

Obtaining Allocation Context CHAMELEON tracks information at the level of an allocation context. This requires that an allocation context be obtained whenever a collection object is allocated. We have implemented two methods for obtaining the allocation context: (i) a language-level method based on walking the stack frames of a `Throwable` object; (ii) a method using JVMTI.

The JVMTI-based implementation is significantly faster than the `Throwable`-based implementation which requires the expensive allocation of a `Throwable` object, and the manipulation of method signatures as strings (our native implementation works directly with unique identifiers, without constructing intermediate objects to represent the sequence of methods in the context).

We are currently working on a third implementation using a modification of the JVM to obtain bounded context information in a lightweight manner. In addition, there are many approaches that target the problem of obtaining context [7, 9, 10, 28], we intend to explore some of these in future work.

Sampling of Allocation Context: To further mitigate the cost of obtaining the allocation context, CHAMELEON can employ sampling of the allocation contexts. Moreover, when the potential space saving for a certain type is observed to be low, CHAMELEON can completely turn off tracking of allocation context for that type. (Technically, sampling is controlled at the level of a specific constructor.)

Available Implementations Our goal in this work is to study the problem of collection implementation selection, and not to improve the default collection implementations. There are many alternative open-source collection implementations [1, 2, 3, 4], varying in terms of robustness, compatibility, and performance. In principle, these implementations can be swapped-in as additional possible implementations for the collection interfaces, with appropriate selection rules on when they should be used.

In our experiments, however, we used our own alternative implementations for collections, for example:

- List:
  - ArrayList - resizable array implementation.
  - LinkedList - a doubly-linked list implementation.
  - LazyArrayList - allocate internal array on first update.
  - IntArray - array of ints. (Similar for other primitives)
- Set (and similarly for Map):
  - HashSet (default) - backed up by a HashMap
  - LazySet - allocates internal array on first update
  - ArraySet - backed up by an array
  - SizeAdaptingSet - dynamically switch underlying implementation from array to HashMap based on size.

Further performance improvements can be achieved by swapping additional implementations under the appropriate conditions. However, some of these conditions are subtle. For example, selecting an open-addressing implementation of a HashMap (e.g., from the Trove collections) requires some guarantees on the quality of the hash function being used to avoid disastrous performance implications. This is hard to determine in Java, where the programmer can (and does) provide her own `hashCode()` implementation.

Context Information As mentioned previously, the `ObjectContextInfo` object collects the usage pattern for collection instances. This information is aggregated into the `ContextInfo` maintained for the corresponding allocation context. As we will see later, with VM support, the context information can also contain information about the heap usage of collections allocated at the given allocation context. As we mentioned already, our design allows us to benefit from VM support, but can also be used when such VM support is absent.

4.3 VM Support

While gathering information at the library level is useful, it is often very difficult to obtain any kind of global view of how collections fit into the whole behavior of the program. For example, even though a particular context allocates memory at a high rate, it is still not clear whether there is much benefit globally in tracking collection usage, for it may be the case that it is a small percent of total memory. Also, it may often be useful to monitor the application with very low overhead, without tracking any library usage, in order to determine whether there is any potential whatsoever in changing the implementation of collections.

One place where much of this global information can be accessed is during the GC cycle. By examining the program heap during a GC cycle, we can calculate various collection parameters such as distribution of live data and collection utilization. Moreover, with careful techniques, this valuable information can be obtained with virtually no additional cost to the program execution time, and as part of normal operation of the collector. To that end, we extended the GC to gather valuable *semantic* information pertaining to collections. At the end of each cycle, the collector aggregates this information in the `ContextInfo` object (which also contains trace-based information). The library can then inspect the combination of trace and heap information at the same time.

4.3.1 Context-Sensitive Collection Data

Note that simply examining the heap is often not enough, especially in large applications with thousands of program sites allocating collections. In particular, we would like to focus on specific allocation sites in the program which have the highest potential for gain. To that end, if the library maintains context information, the collector will automatically take advantage of this and record various context-specific information into the `ContextInfo` object.

4.3.2 Collector Modifications

In our implementation, we used the base parallel mark and sweep garbage collector, which works in the standard way. First, the roots of the program are marked (thread stacks, finalizer buffers,

Name	Description
Live Data	The size of all reachable objects
Collection Live Data	Total occupied size of collection objects
Collection Used Data	Total used size of collection objects
Collection Core Data	Total core size of collection objects
Collection Object Number	Total number of live collection objects
Type Distribution	Live size breakdown for each type

Table 3. Statistics gathered on every garbage collection cycle for each *allocation context*

static class members, etc). Then, several parallel collector threads perform the tracing phase and compute transitive closure from these roots, marking all objects in that transitive closure. Finally, during the sweeping phase, all objects which are not marked are freed. In our system, the number of parallel threads is the same as the number of cores available in hardware. We note that our choice of this specific collector can possibly lead to different results than if we had used for example a generational collector. However, the improvements in collection usage are orthogonal to the specific GC.

We have instrumented the base collector to compute various semantic metrics during its marking phase. The set of metrics computed by collector is shown in Table 3. From these metrics, we can compute aggregate per-context metrics over all GC cycles as described in Section 3.2.2 and shown in Table 1.

Semantic ADT Maps Typically, a collection object may contain several internal objects that implement the required functionality. For example, an ArrayList object may contain an internal array of type java.lang.Object[] to store the required data. This means that if we blindly iterate over the heap, we will not be able to differentiate object arrays that are logically part of ArrayList and those object arrays that have nothing to do with collections (e.g. allocated outside of ArrayList methods). This lack of semantic correlation between objects is a common limitation of standard profilers. Therefore, to efficiently obtain accurate statistics (such as size) about collections, we use what we call *semantic maps*. In brief, every collection type is augmented with a semantic map which describes the offsets that the collector use to find information such as the size of the object (which may involve looking up the size of the underlying array), the actual allocated size and its underlying allocation context pointer. Semantic maps are pre-computed for all collection types on VM startup. Using semantic maps allows us to obtain accurate information by avoiding expensive class and field name lookups during collection operation. Further, because the whole process is parametric on the semantic maps, we can run the system on any collection implementation (including custom implementations).

Operation Every time the collector visits a non-marked object, it checks whether it is an object of interest (a collection object). In that case, it consults the semantic map of its type and quickly gathers the necessary statistics such as the live data occupied by the object (and its internal objects), the used data and the core data (the ideal space if we had only used a pointer array to represent the application data). Further, if the object tracks context information, using the semantic map, the collector finds the `ContextInfo` object and records the necessary information for that allocation context (as described in Table 3).

4.4 Discussion

By augmenting the GC with semantic ADT maps, we were able to automatically and continuously compute various useful context-sensitive utilization metrics specific to the semantics of collections. Moreover, because the statistics are gathered during normal collection operation, no additional performance overhead is incurred.

The information obtained from the collector can be used in various ways. In our case, we propagate the information back to the `ContextInfo` object in the library in order to allow the tool to make a more informed decision by combining this with the library trace-based information. In addition, we also record the results for each cycle separately (it is up to the user to specify what they want to sort the results by as well as how many contexts to show) for further analysis. This information can be readily used by the programmer to quickly focus on contexts which have the most potential for further improvement.

Finalizer Usage In our early versions of this tool, we extended all collection implementation types with finalizer methods. However, we found that finalizers noticeably slowed the system down. One of the main reasons for this is that finalizer objects live for an additional collector cycle and hence all objects transitively reachable from the finalizable object will also live for an additional cycle (even if they are never referred by the `finalize()` method). We still rely on selective usage of finalizers and we use them only for `ObjectContextInfo` objects. These objects are usually very small (few words) and do not have other objects in their transitive closure. Moreover, in the online version, `ObjectContextInfo` objects are not always allocated, further mitigating any costs associated with finalizers. Note that for our purposes, we can also easily compute these statistics in the sweeping phase of the garbage collection cycle (and not rely on finalizers).

5. Experimental Results

5.1 Benchmarks

Because our tool runs on top of a production virtual machine and requires no changes to the application program, we were able to quickly run CHAMELEON on various applications. In our results, we focus on space-critical applications such as SOOT [26], TVLA [18] and FINDBUGS [14]. We also ran CHAMELEON on all of the Dacapo benchmarks [8]. Most of the Dacapo benchmarks do not make intensive use of collections, and hence our tool showed little potential saving for those. However, it did show that there is potential on the benchmarks BLOAT, FOP, and PMD. Hence, we concentrated our efforts on the results for these benchmarks and we present those later in this section. The inputs we used for our benchmarks are an internal Dacapo version for SOOT, TVLA source code for FINDBUGS, the large inputs for Dacapo benchmarks, and an analysis problem—-showing that Lindstrom's binary-search-tree traversal preserves the shape of the underlying tree for TVLA. Also, in our experiments we did not track the potential in benchmarks such as HSQLDB which use their own collection classes. However, with very little manual effort in the library, we can also profile such applications. The collection-aware GC can profile them already as it is parametric in the semantic maps that describe the custom collection classes.

5.2 Methodology

For each benchmark, we took the following steps towards optimizing collection usage:

1. Run CHAMELEON on the application. Based on the results, evaluate whether there is any saving potential. If there is no potential, move on to the next application, otherwise, proceed to the next step.
2. For benchmarks with potential, CHAMELEON reports the allocation contexts in sorted order with the appropriate suggestions.
3. Modify the top allocation contexts using the tool suggestions. This is a replacement step and hence can be easily automated.
4. Repeat steps 1-3 on the modified version.

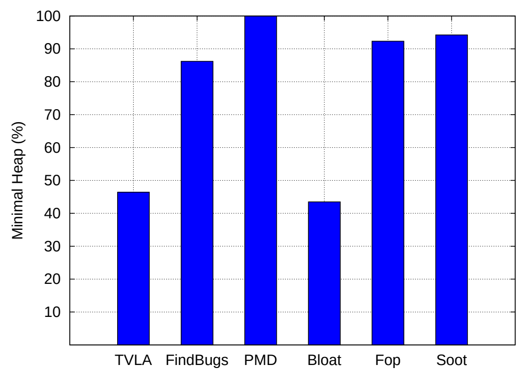

Figure 6. Improvement of minimal heap size required to run the benchmark, shown as percentage of the original minimal heap size.

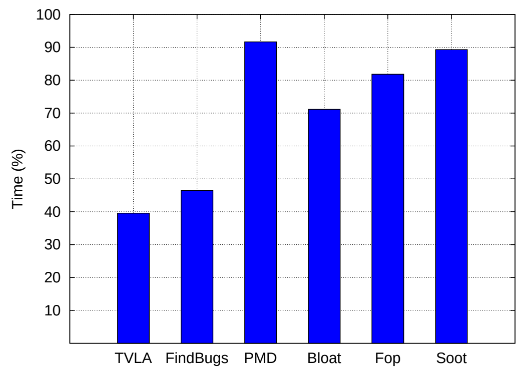

Figure 7. Improvement of running times of the benchmarks after applying fixes suggested by CHAMELEON, shown as percentage of the original running time. Running times were obtained by running each benchmark with its corresponding original minimal-heap size.

5. Compare the gains for the top allocation contexts in the before and after versions.
6. Evaluate the effect of collection optimizations in terms of the minimal-heap size required to run the program, and the execution time when running with the original minimal-heap size.

5.3 Results

Fig. 6 shows the improvement of minimal space required to run the benchmark after applying fixes suggested by CHAMELEON. Fig. 7 shows the improvement of running times of the benchmarks after applying fixes suggested by CHAMELEON. The running times were obtained by running each benchmark with its corresponding original minimal-heap size requirement. Our experiments were obtained on an Intel Xeon 3.8Ghz dual hyper threaded CPUs, 5GB RAM platform running a 64 bit Linux. Next, we discuss each application we considered in more details.

bloat The potential for bloat is shown in Fig. 8. The x-axis is the number of the GC cycle, while the y-axis is the percentage of the total live data computed at the end of the GC. This output is obtained directly from the collection-aware GC. The figure shows that bloat's footprint is dominated by a spike of collections (at GC#656 in the figure), where the true required space for the collections is significantly lower.

The top allocation context reported by CHAMELEON for BLOAT corresponds to this spike of collections, and had a potential

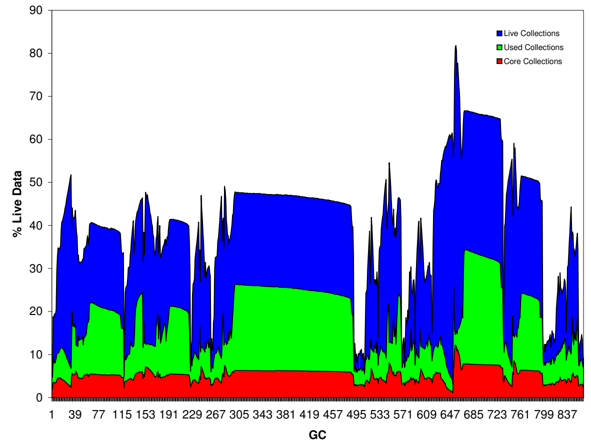

Figure 8. Percentage of collections in original version of bloat

that dominated the potential of all other contexts. Furthermore, CHAMELEON reported that most of the `LinkedLists` allocated at that context remained empty and were never used. Around 25% of the heap at that point of execution was consumed by `LinkedList$Entry` objects that are allocated as the head of an empty linked list. More than 20% of space can be saved by making the lists into `LazyArrayLists`, but a simple manual modification in the code can make the allocation itself lazy, which reduces the minimal-heap size required to run the program by 56%.

FOP In FOP (v0.95), based on the tool recommendations, some `HashMaps` were replaced with `ArrayMaps` and initial sizes of other collections were tuned. There was also one context that allocated collections that were never used (in `InlineStackingLayoutManager`). The result is a 7.69% reduction in the minimal-heap size required to run the program.

Findbugs Based on CHAMELEON suggestions, we replaced some `HashMaps` by `ArrayMaps`, `HashSets` by `ArraySets`, and the initial sizes of other collections were tuned. We also performed lazy allocation for `HashMaps` in contexts where large percentage of the collections remain empty. The overall result is a reduction of 13.79% in the minimal-heap size required to run the program.

PMD PMD was already manually optimized to a correct collection usage. `EMPTY_LIST` was assigned to List pointers when needed and the initial size of many `ArrayLists` was manually set. CHAMELEON discovered many empty and small sized `ArrayLists` that were mistakenly initialized to a high number. We manually performed lazy allocation for these `ArrayLists` which reduced more than 20 million `ArrayList` allocations. In addition, we set the tuned size of lists and replaced `ArrayList` allocation by SingletonList. And also replaced some `HashSets` by `SizeAdaptingSet` (similarly for maps). Unfortunately, all these changes did not reduce the minimal heap size required to run PMD. There are two main reasons for this. The first is that most of the reduced collections are short lived. The second is that most of the long-lived collection data in PMD is large and stable `HashSets` as well as large `ArrayLists`. However, even though our modifications did not reduce the minimal heap size. The number of GCs reduced by 16% which led to a runtime improvement of 8.33%.

Soot SOOT's heap consists of many small objects that are long-lived. It's intermediate representation of program entities makes intensive use of Collection classes. For the most part, SOOT uses `ArrayLists` for flexibility. However, the initial capacity of the lists is rarely provided, and the overall utilization of the lists is rather low (overall, around 25%). For cases in which lists are known to be singletons, SOOT sometimes uses a designated type `SingletonList` to reduce space overhead.

The collection choices we applied in SOOT were simple. Using our tool, we first observed that in the few top contexts in which `ArrayLists` were used to store singletons (by construction), the constructed collections are never modified, and replaced them with immutable `SingletonList` (e.g., in `JIfStmt`). We note that the SOOT team has made a similar selection for other commonly used types. The second suggestion CHAMELEON pointed out is the large potential saving for `ArrayLists` created in `useBoxes` methods. The idiom there is one of aggregation of used values up a tree. Every node creates an `ArrayList` of its uses, and aggregates uses from its children. The result is the creation of many `ArrayLists` that are being rolled into other `ArrayList` using `addAll`. Avoiding all temporaries requires a major rewrite of the code, but even without rewriting the code, we selected proper initial sizes for these lists. The overall result for SOOT was a saving of 6% in space, and 11% improvement in the running time.

TVLA Most of the heap in TVLA is dedicated to storing the abstract program states that arise during abstract interpretation. The abstract program states use collections to store the state information. Most of the collection data is stored in `HashMaps` from seven contexts. CHAMELEON points this collections as ones that can be replaced by `ArrayMaps`. Replacing these collections provides a minimal-heap reduction of 53.95%. CHAMELEON also pointed an initial size setting for several contexts and `LinkedList` that can be replaced by an `ArrayList`.

5.4 Discussion

Experience with Fully Automatic Replacement Our tool can run in fully-automatic mode in which replacement of collections is performed during program execution. Due to the high cost of obtaining allocation contexts, we expected the tool to incur a high time overhead, and only evaluated its effectiveness in terms of space reduction. We ran the tool in the fully automatic mode for all of our benchmarks to evaluate the quality of its replacement decisions. Much to our surprise, for most benchmarks, the overall slowdown was noticeable, but not prohibitive.

For TVLA, the space saving achieved was identical to the one we got with the manual modification. However, the impact on running time was significant, due to the cost of obtaining allocation contexts. Overall, TVLA suffered a slowdown of 35%. For space-critical applications such as TVLA this may be an acceptable tradeoff in practice. The only benchmark for which the slowdown was prohibitive (6x slowdown) was PMD, which performs massive rapid allocation of short-lived collections, which amplified the cost of obtaining allocation contexts.

Our experiments indicate that the performance bottleneck standing in the way to fully automatic replacement is the task of obtaining an allocation context. We believe that with better VM support for this functionality, fully online replacement is within reach.

Iterators In many of our benchmarks, we have observed the (somewhat expected) massive creation of iterator objects. Quite often, the iterators were created over empty collections. For some of the collection interfaces (e.g., Set), the creation of a new iterator object can be avoided in this case in favor of returning a fixed static empty iterator. However, some collection interfaces allow addition of new items through an iterator, and therefore require that a new iterator object will be created even when the collection is empty.

Specialized Partial Interfaces The Java collection interfaces are rather rich, and pose significant restrictions on the underlying implementations. More efficient implementations could be introduced if collection interfaces are minimized, or at least separated. For example, the `List` interface currently supports a list iterator that can traverse the list both forward and backward. For practical purposes, such interface precludes an underlying implementation of using a singly-linked list. While we can leverage static analysis to show place, it seems more desirable to modify the library interfaces to permit additional implementations.

6. Related Work

Recent work by Jones and Ryder [15] suggests that allocation context is indicative of object behavior and argues that GC should take advantage of this (rather than relying on fixed heuristics). We use a similar observation to gather semantic-oriented object metrics and perform corrective actions accordingly.

The challenge of freeing the user from managing and choosing the right data structure for their application is not a new one. For example, in the context of the high-level language SETL, Dewar et. al [12] suggest the usage of a special sublanguage to declaratively specify the type of a data structure that a set or a map of a SETL program should use. A compiler then takes as input the SETL program and the data structure specification in the sublanguage and outputs an efficient implementation. More work on this subject by Schonberg et. al [23] focuses on eliminating the need for manually specifying the structures in a sub-language. It proposes an analysis that takes as input a pure SETL program and automatically infers suitable data structures for it. A similar line of work based on static analysis is presented by Low [19]. In contrast, our work is done in the context of a lower-level language (Java) where the operations on the data-structure (collection) are explicit. Further, our work is centered around dynamic (rather than static) analysis. The mere size of current programs combined with modern language features make it challenging to statically optimize collection usage.

Active Harmony [11] is a system for automatic tuning of programs. The system contains a layer for automatic tuning of parameters as well as a library specification layer that helps the application select the right library to execute. Active Harmony requires each library to provide a performance-evaluation function, and a cost-estimate function. The functions are used by the tuning algorithm to evaluate the performance of the library, or estimate its possible behavior. In contrast, our work targets general purpose object oriented programs where the program is executed in a runtime environment, and optimizes this environment to collect valuable information. In addition, we use allocation contexts to share historical information between objects as well as gain some metric of stability of collection behavior. Moreover, our work combines the GC and VM information per context to decide which contexts are worthy to optimize.

More recent work dealing with the challenges of using custom collections in Java is that of Sutter et. al [25]. They apply static analysis to determine when a replacement of an existing collection type with a custom type is possible without violation of type constraints. Further, they profile several applications to determine where a replacement may be possible. Subject to the type constraints, their analysis automatically replaces existing types with custom types. Their replacement is based on allocation site (rather than context as is in our case). Their profiling information does not include heap information (as we do via VM support). We see our work as complementary. We can provide a more detailed profiling information and then use a static analysis to determine when it is safe to replace one type with another. However, because our system supports wrappers, we are able to always make a replacement as the type safety cannot be violated.

Recently, there has been work on application-specific selection of GCs, see Soman and Krintz [24] for details. The challenges they face are broadly similar to ours: when should one switch from one

GC to another and what application characteristics should switching take into account. For example, the authors describe a scenario of switching to a GC that is tuned for resource-constrained environments when the memory becomes scarce. GC switching occurs at pre-defined points when all application threads are stopped. Switching GCs is complex as it may involve on stack replacement to adjust methods to the specific GC (e.g. use write barriers for generational GC). In our case, switching is localized as it occurs when a collection object is allocated which does not require us to stop application threads. An interesting item of future work is looking into GC strategies that have semantic knowledge of collection objects. For example, the GC may allocate ArrayList and its internal object array together for locality purposes.

Recently, there has been work on semantically modifying the GC to detect various correctness properties, see [5, 6]. In our case, we extended the GC to gather context-specific collection information. We believe that exploring conceptually small but highly beneficial semantic extensions to the VM is a fruitful area of research.

Additional research were done in the field of automatic tuning. A works for automatically tuning linear algebra was done by Whaley and Dongarra [27] in the ATLAS project. Their work automatically generates efficient linear algebra routines for a given microprocessor. They show an automatic generation of matrix multiplication routines for different hardware architectures. A work for automatically choosing a decision heuristic for a SAT solver was done by Lagoudakis and Littman [17]. In their work a decision heuristic is chosen according to a value function, which is calculated on the current state of the search. The value function is created beforehand, using a training set. There are also works on dynamic pretenuring in GC [13, 16]. These works use the same notion as ours of automatic tuning, however, these works do not try to select and manage data structures and tackle different problems in a different domain.

Acknowledgments

We thank Matthew Arnold, Nick Mitchell, Mooly Sagiv, and Gary Sevitsky.

References

[1] Apache collections. http://commons.apache.org/collections/.

[2] Google collections. http://code.google.com/p/google-collections/.

[3] Javolution collections. http://javolution.org/.

[4] Trove collections. http://trove4j.sourceforge.net/.

[5] AFTANDILIAN, E., AND GUYER, S. Z. GC assertions: Using the garbage collector to check heap properties. In *MSPC* (2008), ACM.

[6] ARNOLD, M., VECHEV, M., AND YAHAV, E. QVM: an efficient runtime for detecting defects in deployed systems. In *OOPSLA '08: Proceedings of conference on Object oriented programming systems languages and applications* (2008), pp. 143–162.

[7] BARRETT, D. A., AND ZORN, B. G. Using lifetime predictors to improve memory allocation performance. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation* (New York, NY, USA, 1993), ACM, pp. 187–196.

[8] BLACKBURN, S. M., GARNER, R., HOFFMAN, C., KHAN, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: conf. on Object-Oriented Programing, Systems, Languages, and Applications* (2006), pp. 169–190.

[9] BLACKBURN, S. M., SINGHAI, S., HERTZ, M., MCKINELY, K. S., AND MOSS, J. E. B. Pretenuring for java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2001), ACM, pp. 342–352.

[10] BOND, M. D., AND MCKINLEY, K. S. Probabilistic calling context. *SIGPLAN Not. 42*, 10 (2007), 97–112.

[11] CHUNG, I.-H., AND HOLLINGSWORTH, J. K. Runtime selection among different api implementations. *Parallel Processing Letters 13*, 2 (2003), 123–134.

[12] DEWAR, R. K., ARTHUR, LIU, S.-C., SCHWARTZ, J. T., AND SCHONBERG, E. Programming by refinement, as exemplified by the setl representation sublanguage. *ACM Trans. Program. Lang. Syst. 1*, 1 (1979), 27–49.

[13] HARRIS, T. L. Dynamic adaptive pre-tenuring. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management* (New York, NY, USA, 2000), ACM, pp. 127–136.

[14] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. In *OOPSLA '04: Companion to the conference on Object-oriented programming systems, languages, and applications* (2004), pp. 132–136.

[15] JONES, R. E., AND RYDER, C. A study of java object demographics. In *ISMM '08: Proceedings of the 7th international symposium on Memory management* (2008), pp. 121–130.

[16] JUMP, M., BLACKBURN, S. M., AND MCKINLEY, K. S. Dynamic object sampling for pretenuring. In *ISMM '04: Proceedings of the 4th international symposium on Memory management* (New York, NY, USA, 2004), ACM, pp. 152–162.

[17] LAGOUDAKIS, M. G., AND LITTMAN, M. L. Learning to select branching rules in the dpll procedure for satisfiability. In *SAT* (2001).

[18] LEV-AMI, T., AND SAGIV, M. TVLA: A framework for Kleene based static analysis. In *Saskatchewan* (2000), vol. 1824 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 280–301.

[19] LOW, J. R. Automatic data structure selection: an example and overview. *Commun. ACM 21*, 5 (1978), 376–385.

[20] MITCHELL, J. C. Representation independence and data abstraction. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1986), ACM, pp. 263–276.

[21] MITCHELL, N., AND SEVITSKY, G. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference* (2003), vol. 2743 of *Lecture Notes in Computer Science*, pp. 351–377.

[22] MITCHELL, N., AND SEVITSKY, G. The causes of bloat, the limits of health. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (New York, NY, USA, 2007), ACM, pp. 245–260.

[23] SCHONBERG, E., SCHWARTZ, J. T., AND SHARIR, M. Automatic data structure selection in setl. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1979), ACM, pp. 197–210.

[24] SOMAN, S., AND KRINTZ, C. Application-specific garbage collection. *J. Syst. Softw. 80*, 7 (2007), 1037–1056.

[25] SUTTER, B. D., TIP, F., AND DOLBY, J. Customization of java library classes using type constraints and profile information. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004* (2004), vol. 3086 of *Lecture Notes in Computer Science*, pp. 585–610.

[26] VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), IBM Press, p. 13.

[27] WHALEY, C. R., AND DONGARRA, J. J. Automatically tuned linear algebra software. In *Supercomputing* (1998).

[28] ZHUANG, X., SERRANO, M. J., CAIN, H. W., AND CHOI, J. D. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06* (2006), pp. 263–271.