DYNAMIC DATA STRUCTURE ANALYSIS AND VISUALIZATION OF JAVA PROGRAMS

by

Sokhom Pheng

School of Computer Science

McGill University, Montreal

May 2006

A THESIS SUBMITTED TO MCGILL UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF

MASTER OF SCIENCE

Abstract

For many years, programmers have faced the problem of reading and trying to understand other programmers' code, either to maintain it or to learn from it. Analysis of dynamic data structure usage is useful for both program understanding and for improving the accuracy of other program analyses.

Data structure usage has been the target of various static techniques. Static approaches, however, may suffer from reduced accuracy in complex situations and have the potential to be overly-conservative in their approximation. An accurate, clean picture of runtime heap activity is difficult to achieve.

We have designed and implemented a dynamic heap analysis system that allows one to examine and analyze how Java programs build and modify data structures. Using a complete execution trace from a profiled run of the program, we build an internal representation that mirrors the evolving runtime data structures. The resulting series of representations can then be analyzed and visualized. This gives us an accurate representation of the data structures created and an insight into the program's behaviour. Furthermore we show how to use our approach to help understand how programs use data structures, the precise effect of garbage collection, and to establish limits on static data structure analysis.

A deep understanding of dynamic data structures is particularly important for modern, object-oriented languages that make extensive use of heap-based data structures. These analysis results can be useful for an important group of applications such as parallelization, garbage collection optimization, program understanding or improvements to other optimization.

Résumé

Depuis de nombreuses années, des programmeurs ont éprouvé de la difficulté à lire et à comprendre le code source écrit par autrui, soit pour l'apprentissage ou pour la maintenance. L'analyse de l'usage des structures de données est utile à la compréhension d'un programme et à l'amélioration de la précision des autres analyses.

L'usage des structures de données a été la cible de plusieurs techniques d'analyses statiques. Cependant, ces approches statiques courent le risque d'être moins précises lors des situations complexes. De plus, elles ont le potentiel d'être trop conservatrices dans leurs approximations. L'obtention d'une image précise et claire des activités du tas (heap) pendant la durée d'exécution est une tâche ardue.

Nous avons donc faites la conception et l'implémentation un système servant à analyser dynamiquement un tas. Ceci nous permet d'examiner et d'analyser comment des programmes en Java construisent et modifient leurs structures de données. En utilisant un fichier trace profilé d'une exécution de programme, nous avons construit une représentation interne qui reflète l'évolution des structures de données pendant la durée d'exécution. À chaque modification, nous pouvons analyser ces structures et les visualiser. Nous avons alors une représentation précises des structures de données construites ainsi qu'une meilleure connaissance des comportements du programme. De plus, nous démontrons l'utilisation de notre approche pour faciliter la compréhension de l'utilisation des structures de données pas les programmes, pour avoir une connaissances des effets précises du récupérateur de place (GC) et pour établir une limite sur les analyses statiques des structures de données.

Une compréhension plus profonde des structures de données dynamiques est particulièrement importante pour les langages d'objets orientés modernes dont les structures de données sont basées sur le tas. Ces résultats d'analyses peuvent être utile pour un grand

groupe d’applications tels que la parallélisation, l’optimisation du récupérateur de place, la compréhension de programme ou l’amélioration des autres optimisateurs.

Acknowledgments

This work would not have been complete without the support of many. First, I would like to thank my supervisor Clark Verbrugge for his constant guidance, support and encouragement throughout the completion of this work, as well as for his financial support.

I would like to thank both Clark Verbrugge and Laurie Hendren for providing a nice research environment in the Sable lab, where most of this work has been completed. Additional thanks go to members of the Sable Research group for making the lab a friendly place to be. In particular I would like to thank Ahmer Ahmedani, Grzegorz Prokopski, Chris Goard, Chris Picket, Haiying Xu and Nomair Naeem for providing much interesting discussions as to take my mind of my work. A special thanks goes to Bruno Dufour for developing the *J framework; without this my work would not have been possible.

This research has been funded in part by the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Finally, I would like to give special thanks to my parents, my sister, Oliver Chen and the rest of my friends for putting up with me and for their constant support and encouragement throughout my studies.

Contents

List of Figures

List of Listings

Chapter 1
Introduction

1.1 Motivation

Data structure, heap and *shape* analysis techniques summarize dynamic data connectivity, with the goal of improving alias analysis [GH96], automatic parallelization [HN90], optimizing garbage collection [SKS00], debugging, or as part of a general understanding of program behaviour. Investigation of data structure shape and usage is particularly important for programs which make extensive use of heap data, such as Java and other object-oriented languages.

There are many attempts on data structure analysis, but they are mostly static approaches as dynamic approaches tend to have too much overhead. Static approaches to data structure analysis potentially suffer from overly-conservative approximations, easily induced by temporary data structure inconsistencies during updates and modifications. Dynamic approaches, on the other hand, are either very slow due to the overhead, or not complete as they have to leave out much information in order to be able to work at run-time without slowing the program.

In this thesis we investigate heap data analysis from the perspective of dynamic analysis. Even though our technique is slow and not done during run-time, it gives us a complete picture of data structure properties within Java programs. Using complete traces of Java program executions, we reconstruct the entire program execution bytecode and history of

heap-based data as it is changed through program modifications. That way we can keep track of the program's heap nodes and their connectivity.

For smaller programs this allows for the construction of data structure snapshots and animations, visually illustrating evolution of program data, and also encoding a variety of properties of interest, including shape, age of data, node types, connectivity, and so on. The animation of data structures might be used for many purposes, such as learning and understanding a new algorithm. We show, however, that nice animations are not that easily achievable.

For large benchmarks the results of analyses run at each data structure change are graphed to summarize overall behaviour. This permits larger scale investigations of data structure usage, and using a selection of standard Java benchmarks we demonstrate the extraction and analysis of various data that can extend detailed, runtime heap analysis to reasonably sized programs.

Data on number and size of data structures, their general shape, connectedness and entrypoints, all supply useful information on how programs use dynamic data structures, and we show how analysis of such data can provide insights into program behaviour. This includes aspects of data reachability—we can further examine the extent of and variation in garbage data carried through program execution (GC *drag* [RR96]). A complete tracking of heap data also allows us to determine upper limits on the potential accuracy of a more traditional static, conservative tree/*DAG* (Directed Acyclic Graph)/cycle data structure analysis, under different assumptions of available alias analysis information. Most programs in our study are surprisingly simple with respect to heap usage and our results show that static approaches can be quite accurate, at least for common industry benchmarks.

To further demonstrate the flexibility and utility of our analysis system and approach we also define and implement a less traditional data structure shape analysis based on combinatorial topology.

Data structure analysis is difficult, but worthwhile as it can expose a wide variety of interesting program behaviours.

1.2 Contributions

Specific contributions of our work include:

- We provide a design and implementation of a framework for capturing the complete dynamic evolution of data structures in Java programs. Our system supports various data structure analyses that expose interesting and useful benchmark properties.
- We provide a simple technique for data structure visualization: a series of snapshots that can encode current and historical data structure properties, which is turned into animation to easily see the evolution of program data structures.
- We compare accurate runtime data structure analysis data with that achievable by both optimal and simple static approaches, assuming different levels of alias information. This establishes limits on accuracy for static heap analyses.
- We give and discuss experimental results on the actual data structure usage of a number of benchmark programs, including non-trivial programs in the SPEC JVM98 [SPE98] and JOlden [CM01a] suites.
- We define and test a non-traditional analysis for coarse-grained shape classification of data structures. This analysis demonstrates our system flexibility, but may also be useful for another area such as parallelizing optimization.

1.3 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses background and related work on data structure and dynamic analysis. Chapter 3 describes the general design of our analyzer and the kind of analyses and information we can gather. Chapter 4 describes data representation for both smaller programs in the form of animation and large programs in the form of statistical graphs. Chapter 5 gives analyses results performed on a set of tiny, small and reasonably large benchmarks. Finally chapter 6 concludes this work and suggests future directions for research.

Chapter 2
Related Work

Our approach combines two main techniques, dynamic analysis and shape analysis. These have historically been relatively orthogonal pursuits, and so we discuss them separately in section 2.1 and section 2.2, respectively. Since our work also includes aspects of data structure visualization, related works on visualization are described in section 2.3.

2.1 Shape Analysis

Shape analysis is a term to represent static program-analysis techniques which attempt to determine properties of the heap contents. Shape analysis techniques vary from implementing a whole new language for identifying data structures to summarizing them using specialized graphs.

A frequent, and early approach to identifying data structures is to allow the programmer to provide high-level information through program annotations. Hummel et al., for instance, define static annotations to data structures in order to help the compiler identify opportunities for parallelizing transformations [HHN92]. They have developed an approach for the *Abstract Description of Data Structures* (ADDS) where programmers have a way to describe properties of the data structures to the compiler. It is designed to intuitively and accurately describe pointer data structures possessing a form of regularity by describing their shape and traversal properties.

A similar annotation approach is described by Fradet and Le Métayer, who define a new language annotation that integrates the notion of shapes into the C language [FM97]. The notion of shapes to express properties of data structures is described as context-free graph grammars where modifications are defined as rewrite rules; these rules are used to ensure that the graph structure described by the grammar is preserved.

Many have tried identifying data structure shape without modifying the source code. Ghiya and Hendren show how the conceptually simple categorization of data structures into *tree*, *DAG*, or *cycle* can be sufficient for compiler optimization [GH96]. They introduce the idea of using a *direction matrix* to determine whether a heap-directed pointer has a path to another heap-directed pointer or not, and an *inference matrix* to determine if two heap-directed pointers can access common objects. These matrices are used to perform analysis to estimate the shape of the data structure as a tree, DAG, or cycle graph.

More detailed data structure information can be discovered through various kinds of graph abstractions. Klarlund and Schwartzbach's *graph types* build a representation as a grammar describing data structures having a backbone, such as doubly-linked lists [KS93]. Wilhelm et al. [WSR00] define *shape graphs* to represent structural properties of data structures. These graphs are the result of shape analysis, which they define to be a conservative static program-analysis to determine properties of the heap contents. Using *flow graphs*, the shape graphs are modified to give a conservative representation of heap-allocated data structures every time a program point is executed.

Corbera et al. combines static shape graphs with abstract storage graphs to give a more precise shape analysis [CAZ02]. For that to be possible, they extend *static shape graphs* that were first introduced by Sagiv et al. [SRW98] by adding summary nodes for summarizing different structures and by adding a shared attribute to keep track of cycle links, which give a more accurate representation of doubly-linked graphs. Their techniques were later improved by Navarro et al. by approximating the data structures in a graph combining memory locations having similar patterns [NCA$^+$04]. Their analysis maintains topological information on connections between the different memory locations (nodes) in the data structure. The analysis is also based on *reference shape graphs* used to approximate all possible memory configurations appearing after executing a code statement, where each of those nodes represents memory locations which have similar reference patterns.

Recently, Hackett and Rugina described a way of breaking down the entire shape abstraction into smaller component and analyzing them separately [HR05]. That way, it enables them to do local reasoning about a single heap location instead of doing a global reasoning on the entire heap. To do so, they use two kinds of decompositions to break down the shape abstraction. They first use a "vertical" decomposition to identify points-to-relations between regions. Then they use a "horizontal" decomposition to characterize the state of each single heap location, where the reasoning will be done.

Raman and August [RA05] examine complete dynamic traces to generate abstract shape information; this is a similar general approach to ours, although they concentrate on identifying recursive structures, and derive their information from low-level, binary analysis. They use a technique called *Recursive Data Structure Profiling* to help better understand the dynamic memory behaviour of recursive data structures such as trees. An important step of this technique is to reconstruct shape graphs that were created on the fly during the program execution. It gives two representations for shape graphs. It first collects all information on each object node and reconstructs a *Unified Shape Graph*, where each object node contains all reference pointers to and from it. However this graph turns out to be very large for them to keep track of. Therefore they categorize the edges of the unified shape graph into different instances of recursive data structures and only keep track of those instances instead of all objects. With this, they construct a *Static Shape Graph*, which is basically a summary of the unified shape graph. Contrary to their work, instead of only analyzing recursive data structures, we analyze all data structures that are constructed.

While most work done on shape analysis has been done statically on C code, Bogda and Singh have dome some exploratory work on shape analysis for Java code at run-time [BS01]. They show that by analyzing at run-time, they avoid problems that would arise for static analysis such as dynamic loading and binding. However, the trade-off is the run-time cost and the fact that the analysis must work with incomplete information. Since their analysis is based on call graphs, to address these trade-offs, they build their call graphs incrementally based on the previous modification point call graph to avoid the large overhead. This way they are able to prove that good results are possible, although mainly under repeated execution scenarios.

The different approaches described above show what have been done in terms of shape

analysis. Our work will mainly focus on the concept described by Ghiya and Hendren [GH96] to build our analysis, where data structures are categorized into *tree*, *DAG* or *cycle*.

2.2 Dynamic Analysis

Dynamic program analysis can be performed online, or offline through the analysis of program execution trace files. Given the large resource demands of our precise shape analysis we have focused on the latter technique; many inroads have been made to the former [BS01], however.

Trace extraction from Java programs often relies on the use of the Java's built-in Virtual Machine Profiling Interface (JVMPI), or its new replacement JVMTI (Tracing Interface). Brown et al. describe a framework, STEP, for profiler developers to encode general program trace data in a flexible and compact format [BDE+02]. JVMPI is also used by Dufour et al. in the implementation of *J [Duf04], a tool for dynamic analysis of Java programs used to generate Java program metrics [DDHV03]. Our work here builds on the *J framework.

Similarly to STEP and *J, SEAT (Software Analysis and Exploration Tool) is a trace analysis tool developed at the University of Ottawa to explore large execution traces of methods calls [HLLF05]. Traces are displayed in the form of a tree structure in eclipse that contains a set of auxiliary views, which are used to display different information that can be gathered from the trace files. This approach is a bit different from ours. Although they also analyse trace files, they only focus on method calls, whereas our work focuses on the whole program execution.

A slightly different approach to dynamic analysis is ARE, which is A Reverse Engineering tool that gathers runtime data to analyze the dynamic behaviour of software systems [GOP03]. Instead of analyzing execution trace files, ARE uses run-time data such as parameter and object values. ARE focuses mainly on the analysis of reflective (dynamic) methods, whereas our work focuses on all methods.

The Daikon project from MIT [ECGN99] and the Dynamo project from Indiana University [LD97] both provide online forms of dynamic analysis, differing mostly in usage.

Both projects are based on observing runtime values and invariants to perform diverse analyses and optimizations. The Daikon project uses the information to report properties that were true over the observed period, which can then be used for testing and verification. Dynamo is a compiler architecture that uses the information to do runtime optimizations. The challenges of efficient online dynamic analysis are quite different from our exhaustive approach to trace analysis, but the invariant-based approach may be a useful basis for for determining specific data structure properties.

2.3 Visualization

Visualization is useful in a wide range of fields. It is useful for program analysis and for debugging purpose, but it is even more useful for program comprehension or even for learning an algorithm for educational purposes. We will describe some of the work that has been done in those fields.

Visually representing the heap is an existing concern in many area, and it is used for various purposes. For example, Zimmermann and Zeller use heap visualization to debugging purpose [ZZ01]. Their *memory graph* is used for accessing and visualizing memory contents, where each value in memory is a vertex and each pointer is an edge. That way it can capture the program state as a graph. Printezis and Jones use heap visualization to understand program behaviour in order to design the next generation of garbage collectors [PJ02].

Reiss and Renieris use visualization for program profiling in general to better understand the behaviour of their software [RR05]. They do so by displaying a summary visualization of data gathered from a running Java program, which shows the execution of the program as it occurs, but not in terms of data structures.

Reiss uses visualization to provide high-level program-specific information in real time of Java programs [Rei03]. To work in real time, their trace data have to have minimal overhead. In order to cut down on the analysis time, instead of showing everything the program is doing they decided to break down the program execution into intervals, and only showing a summary for each of those intervals. The information they gather for a

class includes the number of entries, the number of synchronization calls and the number of allocations, and for a thread it includes the time spent in each state.

De Pauw and Sevitsky use visualization to display reference patterns for solving memomy leaks in Java [PS00]. By using reference patterns, which are repetitive execution sequences, they can work with complex structure of data space in a simplified, aggregated form. They look for memory leaks by comparing two snapshots of the program's object population, one taken just before a critical operation and one taken right after.

Another kind of visualization technique for program comprehension is also used in ARE [GOP03], where a diagram of all method invocations that involves an instance of the object of interest is displayed. That way it allows the understanding of how that object is being used.

A very natural way to learn and understand an algorithm is to trace through it; unfortunately most of the time the algorithm is very complex, and we could easily get lost if we attempt to manually trace it. Therefore it would be most welcome if the tracing can be done by a tool. This is what the GANIMAL project tries to do [DGK02]. It is a framework for developing educational software, which provide a powerful set of features. Some of its features include the ability to do a step-by-step execution of an algorithm or a parallel execution of it, to get a visualization of invariants for program points and blocks, and most importantly an online algorithm animation, which is useful to visualize the control of loops and recursion. Following the same line of thought, our simple visualization technique can be used to understand an algorithm in terms of how data structures are modified.

Chapter 3
*J Shape Analyzer

This chapter describes the *J shape analyzer and its features. Section 3.1 describes the *J tool and how the *J shape analyzer fits in. Section 3.2 describes how the *J shape analyzer is is implemented and added to the *J analyzer framework. It also describes how data structures are internally represented and the kind of information that can be encoded with the data structures. Analyses that were implemented are also described as well as some restrictions to the system.

3.1 Background

In order to perform dynamic analysis for accurately finding data structure modifications we need a way to know what exactly happened during the execution of a program: what values were passed, which objects are referenced, and so on. To do so, we use *J, which is a tool for dynamic analysis of Java programs. It consists of two components. The first one is the *J agent, a profiling agent, and the second one is the *J analyzer, an analysis framework. The *J agent consists of a main trace generator that uses the built-in JVMPI (Java Virtual Machine Profiling Interface) to dynamically receive events from a JVM (Java Virtual Machine) that implements that interface; each event is serialized into a single event stream and output to a file called the *trace file*. The *J analyzer is a trace analyzer that reads the trace files produced by the *J agent and perform any number of dynamic analyses on data stored in the trace [Duf04].

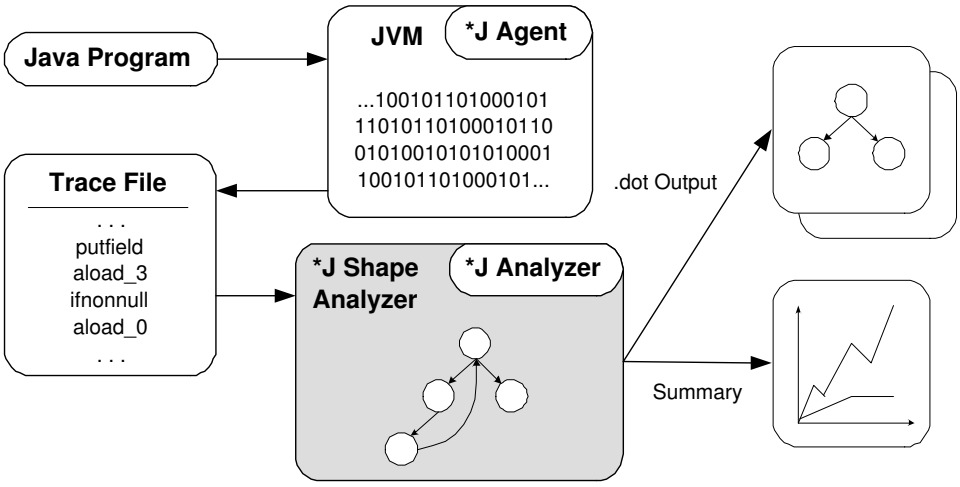


Figure 3.1: Design overview.

The overall flow for the shape analysis system is shown in figure 3.1. The first part of the process is data gathering. Java programs are executed in the JVM where an attached *J agent produces execution trace files of the running program. These trace files are then fed into the *J shape analyzer, which is constructed using the *J analyzer framework. Here the input event trace is used to reconstruct the program data structure and their evolution over time. The *J shape analyzer may apply various analyses such as tree/DAG/cycle analysis, topological shape analysis, etc. The last part of the process is the output representation of the analysis data. Results can be communicated as literal snapshot or animated representations of graph structures, or in the case of larger outputs as graphs of numerical and analysis properties.

3.2 *J Shape Analyzer

For a complete and accurate analysis of runtime data structures, we need complete data on heap objects and references and all values which may be stored in reference fields. *J provides both a complete trace of all instructions executed, and unique identifiers for all objects. We are thus able to reconstruct heap connectivity by tracking which object identifiers are subject and target of reference field writes; this includes reference arrays.

Figure 3.2 shows the flow for the *J shape analyzer. The *J shape analyzer reads events from the generated trace file and processes them one by one. For each event processed, a corresponding update is applied to an internal structure that mirrors the program's heap nodes and their connectivity. This includes the removal of nodes due to GC. At each of these modification points, analyses are then run to determine the evolving properties of the data structure.

Many analyses can be performed on this mirrored data structure, and some of those analyses are explained in section 3.2.4. Apart from the analyses, we can play around with the analyzer's options. One of the options is to be able to set the frequency the analyses are being performed on data structures. The other option is to be able to recognize some data structure properties such as doubly-connected structures. If we did not let the analyzer know it is a doubly-connected structures by explicitly telling it to ignore doubly-connected reference pointers, it will conclude it is a cyclic graph. We can also get analysis result in terms of arbitrary "timelines", including number of data structure modifications or bytecode count.

In the next section we describe how the *J shape analyzer is added to the *J analyzer framework. Then we describe how data structure objects are internally represented, and we follow with a list of information that we are currently encoding about data structures. Section 3.2.4 describes analyses that are implemented, and we end with section 3.2.5 which describes restrictions of the system.

3.2.1 Adding our Analyzer to *J

For the analyzer part of *J to correctly work, since it is a complex tool and contains many different analyses, it needs a way to understand how they work together. Analyses are organized into `Pack`s and `Operation`s, where `Pack` objects are containers for other `Pack`s and `Operation`s, and `Operation` objects contains the actual analysis. All this information is stored in a file called `Scene.java`, which contains the entire hierarchy of `Operation`s and `Pack`s. Furthermore, it contains the main event processing loop indicating which operations and analyses to perform first, second, and so on.

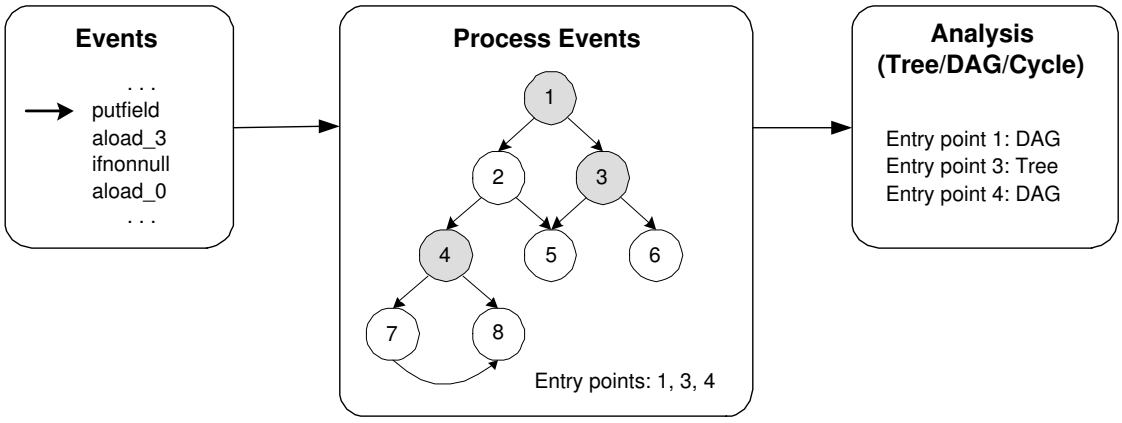


Figure 3.2: *J shape analyzer overview of an analysis.

```
. . .
import starj.toolkits.stacks.StackAnalysis;
. . .
public class Scene {
   . . .
   private void populate(Container container) {
      . . .
      Pack toolkits_stacks = new Pack("stacks",
                      "Contains stack-related operations");
      toolkits.add(toolkits_stacks);
      toolkits_stacks.add(new StackAnalysis("stack",
                     "Output results of stack operations");
      . . .
   }
   . . .
}
```

Listing 3.1: Register the new analyzer in `Scene.java`

In order for the *J shape analyzer to be a part of *J, we first need to register the new analyzer, which we called `StackAnalysis`. We do so by adding a few lines in `Scene.java` as in listing 3.1. We first need to create a new `Pack` object to contain our analysis with a name and a small description of it. Then we have to add the created `Pack` object to the `toolkits` package. Once that is done, we add our analysis to the `Pack` object we just created with the analysis name and its description.

Now that our analysis is registered, we need to follow a certain structure predefined within the *J analyzer framework for it to work properly. The structure is shown in listing 3.2. All analyses have to extend `AbstractOperation` and need to have these three methods: `init()`, `apply(EventBox box)` and `done()`. `init()` is where the analysis starts when the *J analyzer starts. This is where we can initialize objects for the analysis before it actually starts. `apply(EventBox box)` is where events are processed. Whenever there is an event to be processed, this method of each analysis is called to process it. `done()` is called to wrap up the analysis once the end of the trace is reached. It is usually used to print out the result of the analysis or the amount of time it took for the whole analysis to process. In addition, we included two more methods to the analyzer, `operationDependencies()` and `eventDependencies()`. The first method states which other operations provide information used by this operation. Since we need to know the particular bytecode associated with each executed instruction event, we need to make sure that this information will be available to our analysis when it executes. It is done through `InstructionResolver`. The second method is for registration in order to receive the required events. In our case, we were only interested in instruction events, which are called `INSTRUCTION_START` events; that is why we only registered to receive those events. If we wanted other kinds of events, it is the place to register them.

3.2.2 Data Structure Internal Representation

In order to construct an internal data structure representation, we model the complete execution of each thread by interpreting bytecode events. The way we build our system follows somewhat the Java Virtual Machine Specification [LY96]. From figure 3.3, we can see that an `InvocationStack` object is created for each new thread the analyzer

```
package starj.toolkits.stacks;

public class StackAnalysis extends AbstractOperation {
   public StackAnalysis(String name, String description) {
      super(name, description);
   }

   // State which other operations provide info used by this operation
   public OperationSet operationDependencies() {
      OperationSet dep_set = super.operationDependencies();
      dep_set.add(InstructionResolver.v());
      return dep_set;
   }

   // Register to receive required events
   public EventDependencySet eventDependencies() {
      EventDependencySet dep_set = new EventDependencySet();
      dep_set.add(new EventDependency(Event.INSTRUCTION_START,
               new TotalMask(Constants.FIELD_RECORDED), true));
      }
      return dep_set;
   }

   // This is where the analysis starts when the *J analyzer starts
   public void init() { . . . }

   // This is where events are processed
   public void apply(EventBox box) { . . . }

   // This method is called to wrap up the analysis once the end of the trace is reached
   public void done() { . . . }
}
```

Listing 3.2: Structure for the new analyzer to work properly with *J.

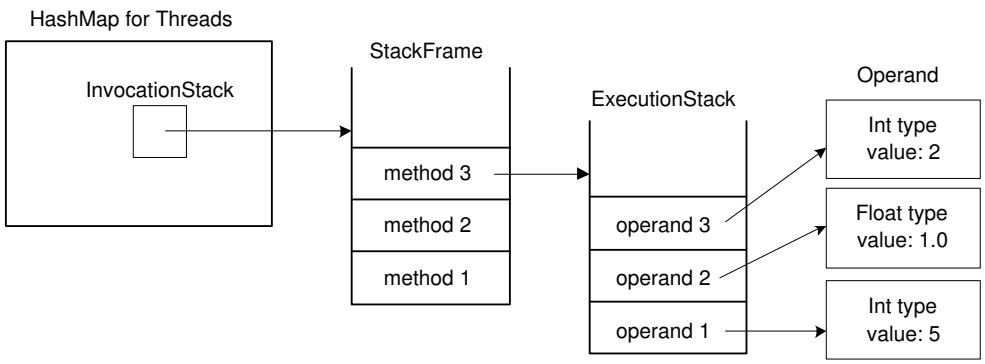


Figure 3.3: Description of the internal representation of an execution context.

encounters, which contains the thread id as well as a `StackFrame` stack object. Each `StackFrame` object is created at the start of every method, which contains the method id, the method signature, info on the local variables and the method parameters as well as an `ExecutionStack` object. Each `ExecutionStack` object contains a stack for `Operand` objects. Each `Operand` object contains information such as whether it is a reference type object, an int type object, a float type object, and so forth. With that, the `ExecutionStack` can mirror the java execution stack so that bytecode from the trace file can be processed.

Most programs contain more than one thread even if the program is not intentionally multi-threaded. That is due to the fact that there are usually many threads created by default by the JVM. Therefore, we need a way to identify which thread is being executed. Fortunately, this is already being handled by the *J tool, which give a unique id for each thread, method and object. Therefore, when an event has a thread id that is different for the previous event, the corresponding `InvocationStack` object is loaded along with the top element of its `StackFrame`, which constitute the method it left off before the other thread was executed.

As events are processed, if the analyzer gets an `INVOKEINTERFACE`, an `INVOKESPECIAL`, an `INVOKESTATIC` or an `INVOKEVIRTUAL` event, all objects from the stack that are needed for the invoked method parameters are loaded to that method's locals. At the method entry, a new StackFrame is created and pushed into the `InvocationStack` object for the thread id. As `INSTRUCTION` events are processed, elements from the

`ExecutionStack` are popped or pushed depending on the bytecode it is processing. For example, if it gets an `IADD` instruction event, two elements from the `ExecutionStack` will be popped, where both of them are of integer type. The value will then be computed before being pushed back to the stack. When the method exits, its `StackFrame` is popped from the `InvocationStack`, and if there is a value to be returned, it is then pushed into the invoker's `ExecutionStack`.

Once we have the system described above set up, we can start mirroring data structures. So far, we have dealt with how events and bytecodes are being handled and processed. However, our goal is to know how objects are connected to each other. Obviously, this only concerns reference typed objects; for each reference typed object or object allocated, we introduce an object called `ObjectAlloc`, which holds a space for reference information on arrays or fields it points to, and also references from other objects. That way when we have a `PUTFIELD` instruction for example, we can store the field object at the corresponding field index. Thus when an analysis is to be performed, a depth first search starting at all roots of the data structure can be done to determine object connectivity and reachability.

3.2.3 Data Structure Properties

From the mirrored representation of the program data structures we are able to find and show a variety of interesting and useful properties. Certainly type, or other node information can be easily included in any graphical representation. We can further encode complex, historical node properties such as relative age of its component nodes, and the data structure can also be examined more abstractly, e.g., in terms of reachability.

Node type in our representations is shown textually. However, since we are most interested in application objects, we distinguish application from library objects through colour as well, and this strategy can obviously be extended to many node properties. Figure 3.4 shows an example of this division, as well as a visualization of the *aging* property: as an object ages, meaning that it lives longer within the program, its colour becomes darker (in figure 3.4 this is applied only to application objects, not library objects). Observing age and type can be a useful way of understanding how a structure is constructed; in figure 3.4, for example, it is evident that the data structure is mostly built bottom-up, with application

nodes near the tree root younger than nodes deeper in the structure.

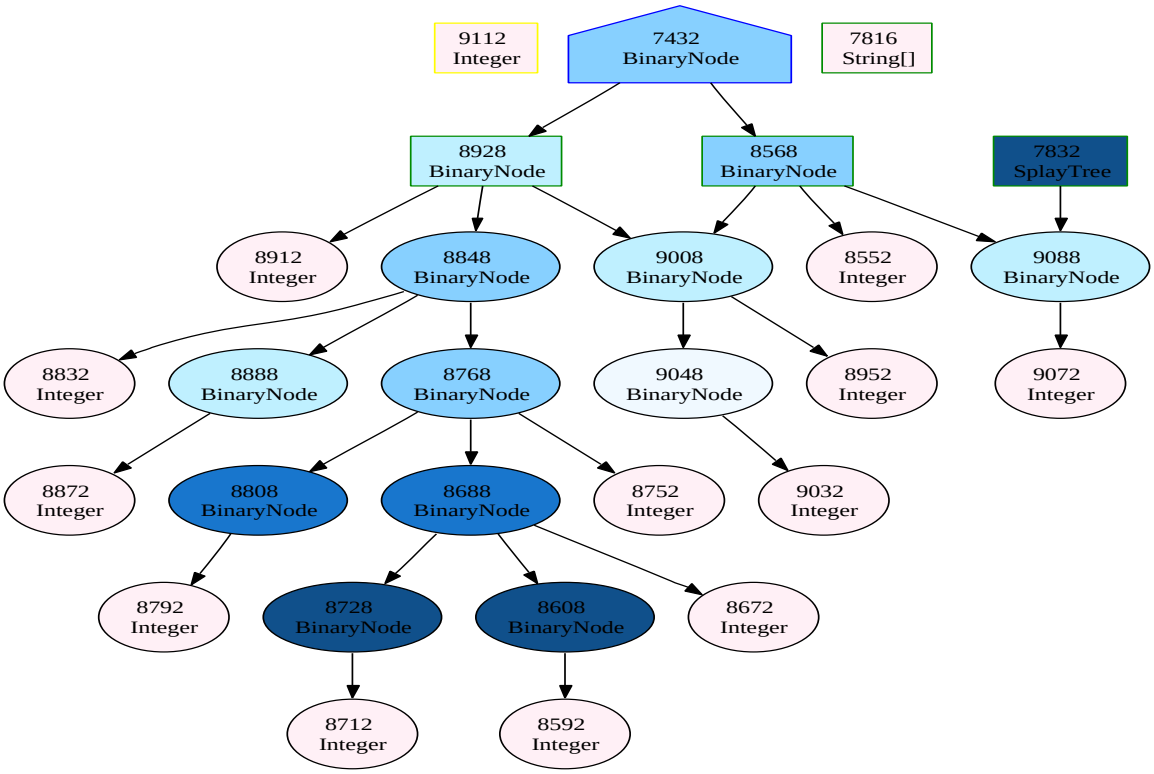


Figure 3.4: A data structure showing the aging property. Nodes are coloured according to their age (and type); all leaf nodes here are library objects, and all internal nodes application objects.

There are many options with aging with which we can play. First, we can age an object every time there is a modification done on the data structures or we can age it at every bytecode execution. In our work, we choose to age objects at every bytecode execution. Second, we have a choice on the colour selection and the range of age for each colour. In our case, colours starting with light to dark are assigned to different range of ages, with the exception of new objects which are of a totally different colour since we want to emphasize the introduction of new objects. The colouring is done in following way:

Age	0	bytecode executions (new object)
Age	1-10	bytecode executions
Age	11-100	bytecode executions
Age	101-1,000	bytecode executions
Age	1,001-10,000	bytecode executions
Age	10,001-100,000	bytecode executions
Age	100,001-1,000,000	bytecode executions
Age	$>$ 1,000,000	bytecode executions

For the moment our tool does not have a way to automatically normalize the intervals for the age classes. It is pre-determined by the user.

Reachability in our system is easily determined. By tracking all object references we also know the set of all root objects, or entry points to the structure. Root objects include static variables, live local variables, and live method parameters. Thus by comparing the transitive closure of references with the set of all allocated but currently uncollected objects we can determine the set of dead objects, not reachable from the root set. This information can be visualized, showing the exact amount and (remaining) connectivity of dead, garbage objects the heap contains. Figure 3.5 shows a visualization of a data structure containing garbage data. Dead objects are drawn with dotted lines, and we can easily see how many there are and exactly how they are connected to each other and to the rest of the structure. Understanding how much data is carried in this way can be useful for garbage collector optimization [RR96].

3.2.4 Analyses

The *J shape analyzer has all necessary information to support the implementation of various analyses, including different summary and shape graph approaches, topological shape analysis, etc. We have implemented a basic tree/DAG/cycle analysis as a proof of concept, and also to investigate the quality and utility of this simple categorization. As a more complex and non-traditional analysis, we have also implemented an analysis based on the combinatorial topology of surfaces to classify the different types of data structures.

There are many ways to look at data structures. Most often they can be viewed as a

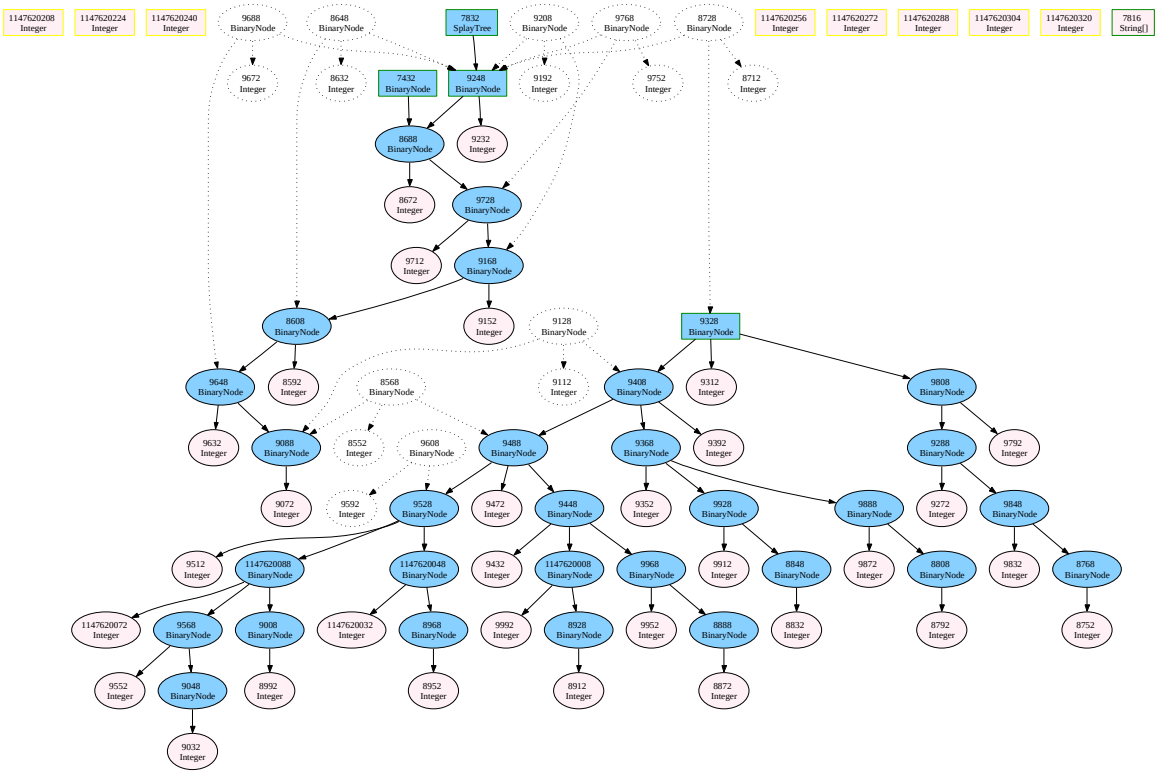


Figure 3.5: Showing garbage nodes in the data structure. Here unreachable nodes are drawn in dotted lines.

whole or according to the reachability of root objects. For the first analysis that we will describe, we use the view given by the reachability of root objects, or entry points.

Tree/DAG/Cycle

Dynamically, a tree/DAG/cycle categorization is quite trivial to compute. From each entry point we simply do a depth-first search to determine whether the nodes reachable from that entry point represent a tree, a DAG or a cyclic graph. This information is then encoded in the graphical output; if the reachable nodes form a tree then the entry point is drawn as a rectangle, if the structure is a DAG then the entry point is drawn as a "house shape" (pentagon), and for cyclic structures a hexagon entry point is used. By performing this analysis at each structure modification we obtain an evolving view of the data, at least in terms of tree/DAG/cycle composition.

This process has one important practical caveat: single, unconnected nodes are considered trees. While this is true in a technical sense, many programs make extensive use of single node objects, and this obfuscates any understanding of more realistic tree usage. For this reason we actually make use of a 4-way categorization, with single nodes distinct from trees.

Connectivity

Note that a given data structure may appear differently from different perspectives: it is common to think of data structures as connected graphs, but analysis information can be distinct for each entry point (reference variable), or generic to the entire connected data structure. Figure 3.4 shows examples of distinct tree and DAG entry points into the same connected structure. In most of our work we use entry point information as fine grain data; connected data structure information, however, is also determined.

Purity

In order to measure the potential accuracy of a static analysis of the same program, we also define a *purity* metric on all data structure references.

Definition Let "$\sqsubseteq$" be a partial order on data structure shapes; e.g., tree $\sqsubseteq$ DAG $\sqsubseteq$ cycle. If the shape computed from a particular reference r at each heap change forms a sequence $s_0, s_1, \ldots, s_n$, then r is *pure* if $s_i \sqsubseteq s_{i+1}$ for all $i = 0...n-1$.

Data structure purity is meant to capture the relative ability of a static shape analysis to accurately determine shape. If despite any changes the data structure is perceived to have the same, constant shape then static analysis may be able to give an accurate shape designation. If, however, the data structure shape changes then any static shape result is necessarily an approximation. Of course data structures are built incrementally—all data structures evolve from trees (single nodes). To avoid considering nearly all DAGs and cycle references as impure we categorize references that never progress downward in shape order as pure. Purity thus over-approximates the accuracy of a static approach.

Based on the above, we compute two measurements on our runtime data. The entry point purity determines purity for each runtime reference. This provides a rough upper bound for static approaches, corresponding to the presence of perfect alias (points-to) information. Less than perfect alias information implies a need to merge information for multiple entry points, necessarily reducing, or at least not improving accuracy.

In the absence of good alias information, a static shape analysis can minimally separate references according to the static class type. To see how well even such a simple approach can determine data structure shape we compute a type-based purity metric; here, shape data for runtime fields with the same static signature are merged together. Purity is then determined from changes in the merged entity.

We must note, however, that entry point purity and even type-based purity may vary at different program statements. Static approaches can and will model these changes, so their accuracy may be greater than our model and what our data suggest.

Combinatorial Topology

Here we present a non-traditional approach for analyzing shapes to demonstrate the flexibility of our system.

Combinatorial topology is the branch of mathematics concerned with essential properties of *shape*. Algorithms exist in combinatorial topology to compute a number of different

qualities on surfaces. We have applied a simple algorithm for computing a canonical shape representation [Jam55]. The algorithm is used to describe general surfaces in canonical form. It decomposes a surface into small pieces, which are then described in terms of equations. The analysis then outputs a 3-tuple to describe that surface. This decomposition and analysis is interesting since it is a non-traditional analysis to be performed on data structures. The result can also be used to describe data structure connectivity and how it can be separated, which can be useful for parallelization.

The algorithm takes a set of equations describing a surface and outputs a 3-tuple describing it. In order to get equations for the surface, we need to cut and unfold it; by retaining the way the surface has been cut and its direction, we get the set of equations. Lets take a look at the example shown in figure 3.6 (a). It shows two pieces of a surface that has been cut along *a*, *b*, *c*, *d*, *e* and *f*. By going clockwise around each pieces and having $^{-1}$ for paths that go in the opposite direction, we get these equations:

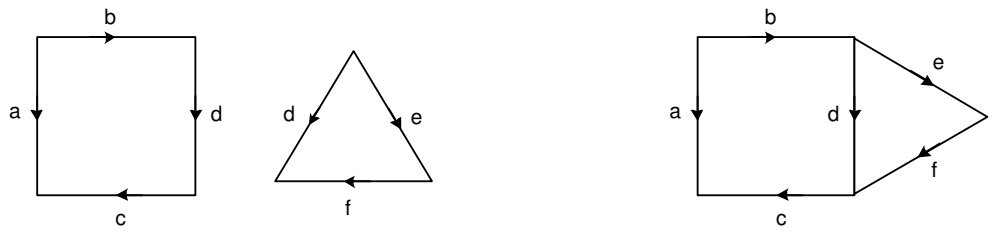
$$\begin{array}{lrcl} \text{Rectangle piece:} & a^{-1}bdc & = & 1 \\ \text{Triangle piece:} & d^{-1}ef & = & 1 \end{array}$$

We then need to reduce these equations to a single equation. Since the *d* path in the rectangle piece and the *d* path in the triangle piece are the same path, these paths can actually be pasted together to produce the surface shown in figure 3.6 (b). The new surface is represented with this equation below:

$$a^{-1}befc = 1$$

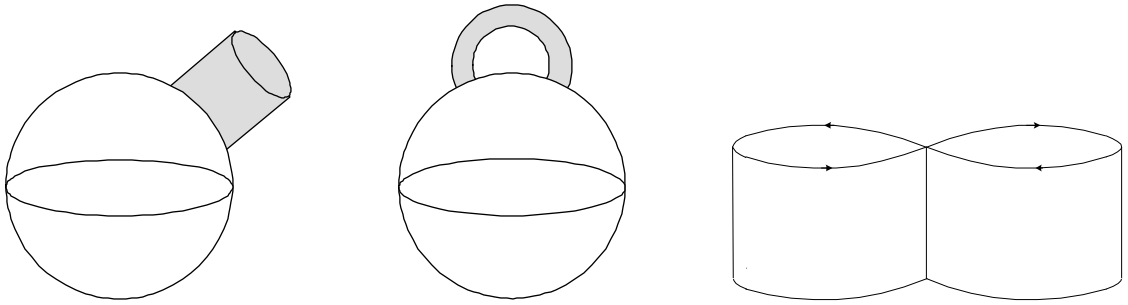
Reduction of the combinatorial equation can then be applied to generate values for each field of the 3-tuple. This algorithm is described in detail in [Jam55].

The 3-tuple fields consist of the number of *cuffs*, *handles* and *crosscaps*. A *cuff* can be seen as in figure 3.7 (a) as being a cylindric form with 1 side filled, where the empty side is connected to a surface. This figure is only a representation of a cuff, which is basically an unconnected edge; it is not specific to cylinders. A *handle*, as its name says, forms a kind of surface with a hole in the middle where we can use it as a handle as in Figure 3.7 (b). Finally, a *crosscap* is a surface where each side of it crosses each other as in figure 3.7 (c). The equations below show an example of a cuff, a handle and a crosscap.



(a) (b)

Figure 3.6: (a) shows an example of surface paths and (b) shows the pieces being pasted together.

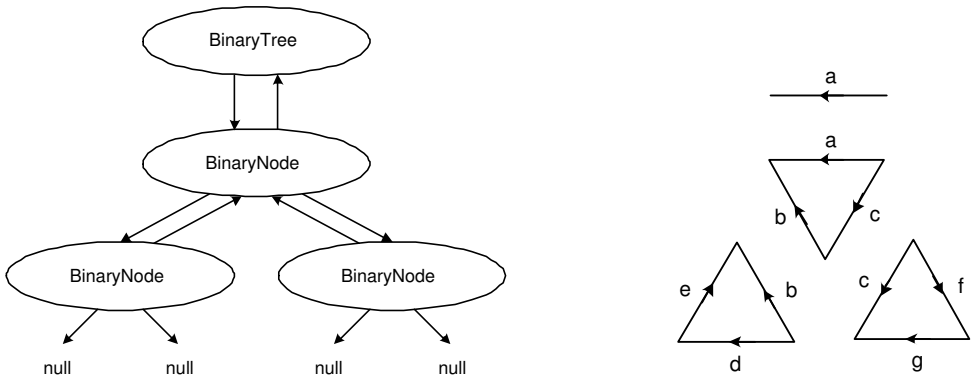


(a) (b) (c)

Figure 3.7: (a) shows an example of cuff in gray, (b) shows an example of handle in gray and (c) shows an example of crosscap.

Cuff:	$ded^{-1}z^{-1}$	$=$	1
Handle:	$aba^{-1}b^{-1}x{-}1$	$=$	1
Crosscap:	ccy^{-1}	$=$	1

To see how a data structure can be mapped into a set of equations fit to be used in the combinatorial topology algorithm, lets take a look at figure 3.8 (a). The figure show a representation of a binary tree with an object pointing to the root of the tree. In order for the concept to work with data structures, we need to assume that all data structures are doubly-connected. Thus if we have a reference pointer from the parent to the child, there is also a reference pointer from the child to the parent. By representing each reference pointer as a path with downward arrows being forward paths and upward ones being backward paths, we get the surface pieces shown in figure 3.8 (b) and the set of equations shown below:



(a) (b)

Figure 3.8: (a) shows an example of a binary tree and (b) shows the surface mapping.

Root pointer:	a	$=$	1
Root object:	$a^{-1}cb$	$=$	1
Left child:	$b^{-1}de$	$=$	1
Right child:	$c^{-1}fg$	$=$	1

The pieces can be pasted together where paths are the same, but opposite direction. By doing so, we get this equation:

$$fgde = 1$$

The analysis result states that the resulting surface is a surface with one cuff.

Apart from the 3-tuple result, the algorithm also outputs a very interesting value called the *Betti number B*. This number indicates the maximum number of cuts that we can be made on a surface without dividing it into separate pieces. Data structure partitioning is important to parallelization, and so computing the Betti number for a data structure may be useful for parallelizing optimizations.

The combinatorial topology analysis tends to produce a coarse categorization, where many different data structures have the same number of cuffs, handles and crosscaps as well as the same Betti number. The algorithm also has the disadvantage of only working on doubly-connected data structures. This greatly limits practical applications of a combinatorial topology approach. Our design here is sufficient to provide a proof of concept of

a non-traditional, non-trivial analysis. Further effort on improving this technique is left for future work.

3.2.5 Restrictions

A few significant restrictions to our approach are implied by the use of the JVMPI interface. We note that the amount of data that can be acquired through it in *J is limited.

- Early events that occur in the virtual machine during start up are not available since they occur before the JVMPI is initialized. Therefore we cannot reliably analyze those events.
- The JVMPI interface only detects events from code written in Java, and thus data from native method executions is not reliably delivered. This can make it difficult to analyze a program precisely—although object allocations and field changes are reported, even from native methods, primitive numerical values are not. Array indexes, therefore, if they come from native methods, are not always known. Fortunately this problem is rarely encountered and does not occur in our benchmark suites.

In our investigations we have restricted our analyses to application code and not include the start up part (JVM start up) in order to ensure we have a complete event trace with minimal information loss due to native method calculations.

Chapter 4
Visualization

Visualization is useful in many fields as described in section 2.3. In our case, we use visualization to look at the evolution of data structures in Java programs. We have two main ways to represent the data gathered. Section 4.1 describes the first way, a literal representation as a series of snapshots, and also describes how the animation of data structure changes is made. Section 4.2 describes the second way, which consists of giving a numerical summary of the data gathered.

4.1 Literal Representation & Animation

The most obvious and direct representation of data structure evolution is as series of literal snapshots of the encoded data structures, as in figures 3.4 and figure 3.5. A snapshot is generated at every update performed on the data structures. By looking at the series of snapshots, we can see how it changed over the program execution. Moreover, we can see data structure properties such as the age of each node at each update and the accumulated dead objects as described in section 3.2.3.

This kind of representation is suitable for small tests, examinations of specific components, and for pedagogical pursuits, but unfortunately is not feasible as a general approach in most benchmarks. The large data sets that must be manipulated in the context of the analyzer impose strong constraints on the style of presentation, and also on the kind of data that can be gathered.

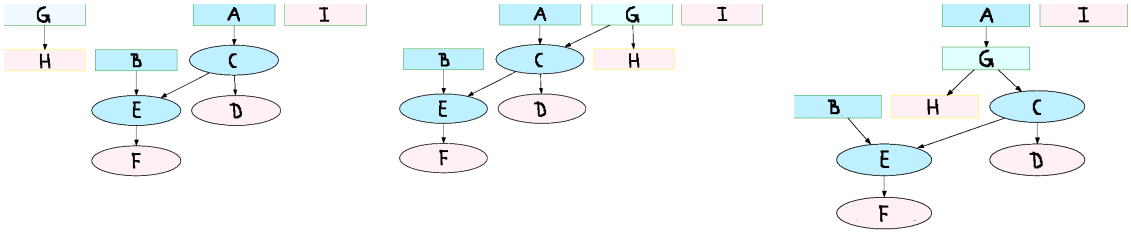


Figure 4.1: SplayTree snapshots. An existing pair of nodes (tree node and associated data) is inserted just below the root of the tree.

Tiny, test programs modify data structures only a relatively small number of times. More realistic programs, however, can perform a very large number of updates; the Jess benchmark from SPECjvm98, for instance, performs more than 48 million heap modifications. Examining all these snapshots is unrealistic for humans. For those programs, instead of generating snapshots for each modification we therefore only generate a snapshot every nth change, for different n depending on the scale of investigation required. This can also help in reducing the computational cost of the analysis.

Snapshot animation itself is surprisingly difficult, even with external tools. In order to have a nice animation of the snapshots, we need to be able to incrementally add/subtract nodes and edges to an existing drawing while ensuring existing nodes and edges do not move. This preserves the location of nodes between snapshots, making node identity trivially obvious as frames change. Current open source and commercial tools for graph layout, however, focus on optimal, static representations, and do not in general attempt to locate nodes in the same place between drawings. This results in animation frames where graphs in successive frames may bear little visual relation to each other, and thus are not useful as a visual replay of data structure behaviour. An example is shown in figure 4.1.

Section 4.1.1 describes tools that were investigated to draw the animations and issues encountered for each of them. Section 4.1.2 propose a way to resolve the animation issue.

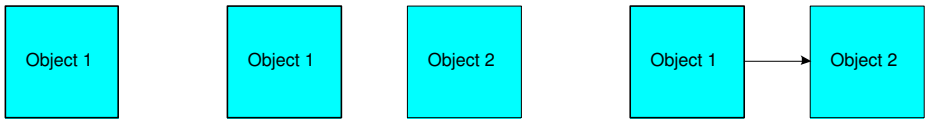


Figure 4.2: Example of what we want of an incremental drawing using Tom Sawyer.

4.1.1 Tools & Issues

In this section, we will show the drawing tools for graph layout that we tried, which range from commercial tools to open source. We will also describe how these tools do not give the results we are looking for. The tools we have investigated include *Tom Sawyer* [tom], *yFiles* from *yWorks* [yFi], and *Neato* and *Dot* from *Graphviz* [GN00].

Tom Sawyer Software

Tom Sawyer is a commercial software for graph visualization, layout, and analysis systems. It is a tool developed in Java that enables development of graph analysis applications quickly and efficiently. The main point that got us interested in this tool is their claim to be able to do incremental drawing.

We used this tool on a very simple example to see if indeed it really draws incrementally the way we want. Figure 4.2 shows how we want the graph to be drawn, but figure 4.3 shows what we get out of the software. The tool still tries to deliver an optimal drawing, which moves objects around. Thus, this tool definitely does not deliver the expected result for our purpose.

Tom Sawyer's incremental drawing works beautifully if, for example, you have two large disconnected graphs, and you want to connect those two graphs together. It will draw the resulting graph such that you will be able to still recognize those two previous graphs, although nodes will still move around. However it is not made for the kind of incremental drawing we want, where we want nodes to remain at their initial drawn location for as long as possible, ideally throughout the whole graph animation.

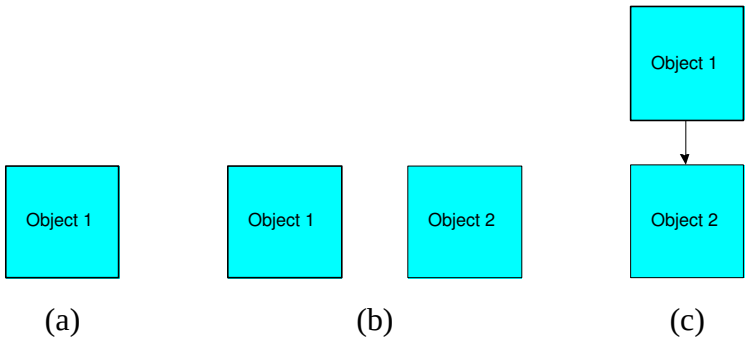


Figure 4.3: Actual result using the Tom Sawyer Software.

yFiles from yWorks

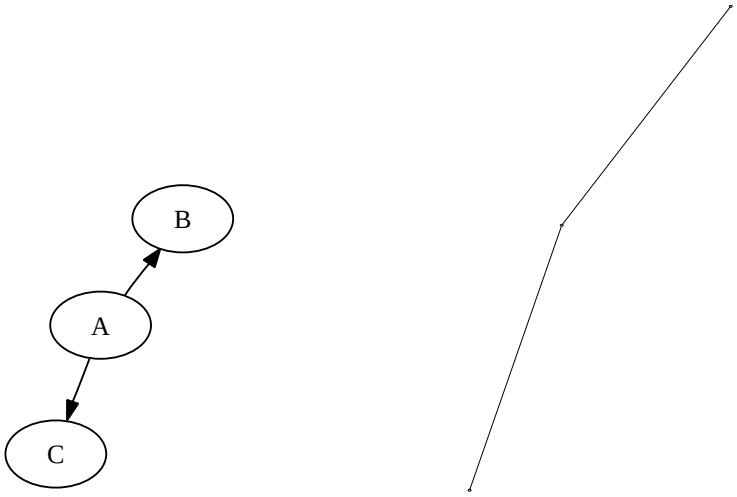
yFiles from yWorks is another commercial software. It contains an extensive Java class library that provides algorithms and components to help better analyze, view and draw graphs. Furthermore it also claims to do incremental drawing.

We use this tool on the same example as with the Tom Sawyer software. Unfortunately, the result was not much different from figure 4.3. The tool is still trying to deliver an optimal drawing moving objects around despite the fact that the incremental option was enabled. Overall, this tool is somewhat similar to the Tom Sawyer software. Note that although these tools do not serve our purposes, it does not mean they are bad for other purposes.

Neato from Graphviz

The Graphviz package is a freely available package of graph drawing programs. A feature of *Neato*, one of the provided graph drawing programs, is to allow nodes to be "pinned". By pinning we mean that objects are glued to the canvas and cannot be moved once they are drawn.

Figure 4.4 (a) shows a simple example of a binary tree with a root and 2 children. Once we enable the pin down option available in Neato, we get the result shown in figure 4.4 (b). Note that figure 4.4 (b) is reduced considerably, 2% of the actual size. For such a simple example to have such a large graph is not ideal. Thus Neato does not solve our problem either, although it may in the future once these bugs are fixed.



(a) (b)

Figure 4.4: Example using Neato: (a) without the pin down option, and (b) with the pin down option.

4.1.2 Resolution

It is clear that existing graph drawing tools are not designed for the kind of incremental animation frames we need to draw. However, it is possible to adapt them to our goals. The basic idea is to build the graphs backward, from finish to start; and below we give a complete and simple algorithm. We still require a separate program for drawing the graphs. For that, we could use any static graph drawing. We chose to use *Dot* from *Graphviz* [GN00] because it uses a simple grammar, draws nice graphs and it is open source.

The algorithm works as follows. With the file containing the complete series of snapshot of the program execution written in *dot* format, we do a first pass on that file to determine every node/object and every edge that is present in the program. Once we determine that, we do a second pass on the *dot* file to determine which node/object and edge are truly part of that snapshot. We then generate a new *dot* file which contains all the nodes and edges that we stored, but everything that was not part of the original snapshot have their node colour, font colour and edge colour set to the background colour in order to make them invisible. In fact, all the snapshots are the same, but what differs is which nodes are visible and which are not. Figure 4.7 shows snapshots which contains visible and invisible

PROCESSINFO($inputFile, outputFile$)

input: $inputFile$ written in dot format containing graphs.
$outputFile$ to write back the modified graphs ready for animation.

▷ first pass, store information
create HashMap $allNodes$ and $allEdges$
while not end of $inputFile$ **do**
for each $line$ in $inputFile$ **do**
if $line$ describes a node **then** store $line$ in $allNodes$ with the object id as key
else if $line$ describes an edge **then**
store $line$ in $allEdges$ with the both objects id connected by the edge as key
end if
end for
end while
writeInfo($inputFile$, $outputFile$, $allNodes$)

Figure 4.5: Algorithm for backward visualization, first pass (process and store information).

WRITEINFO($inputFile, outputFile, allNodes$)

input: $inputFile$ written in dot format containing graphs.
$outputFile$ to write back the modified graphs ready for animation.
$allNodes$ hash map containing all nodes.

▷ second pass, write back to file
create HashMap $visibleNodes$
while not end of $inputFile$ **do**
for each $line$ in $inputFile$ **do**
if $line$ indicates the beginning of a graph **then**
for each object stored in $allNodes$ **do** write node information to $outputFile$
end for
else if $line$ describes a node **then** store the node information in $visibleNodes$
else if $line$ describes an edge **then** write the edge information to $outputFile$
and store the edge and nodes information in $visibleNodes$
else if $line$ indicates the end of a graph **then**
for each node not in $visibleNodes$ **do**
write the node information to $outputFile$ to be drawn as an invisible node
end for
for each edge not in $visibleNodes$ **do**
write the edge information to $outputFile$ to be drawn as an invisible edge
end for
end if
end for
end while

Figure 4.6: Algorithm for backward visualization, second pass (write back to file).

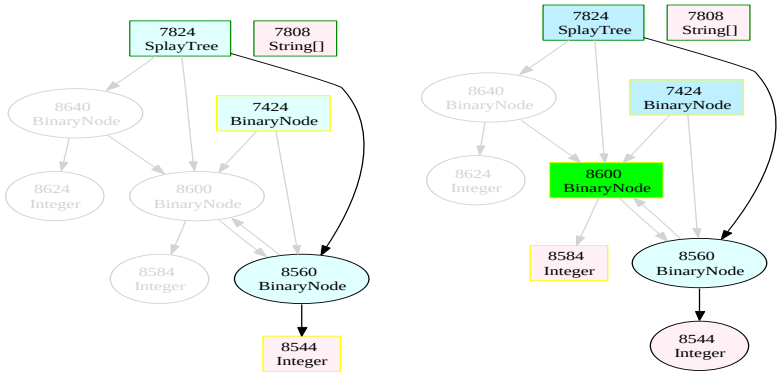


Figure 4.7: SplayTree snapshots showing invisible nodes in light grey.

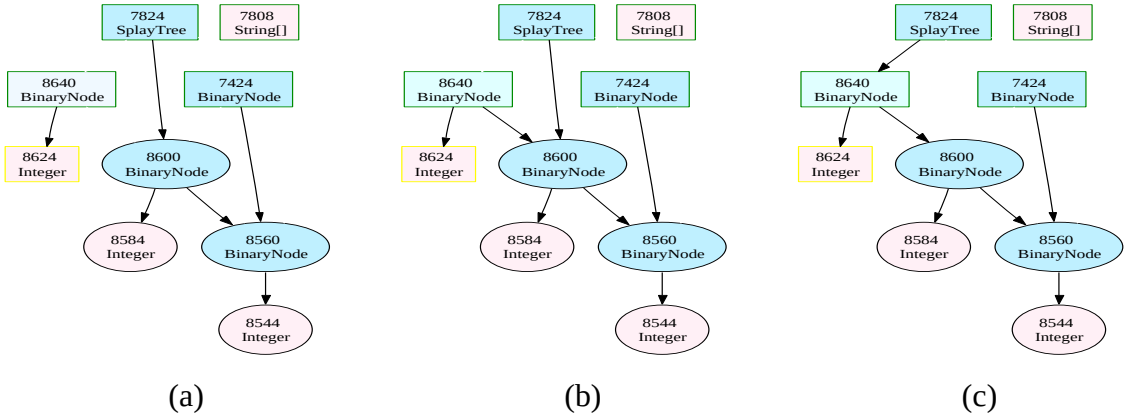


Figure 4.8: SplayTree snapshots with incremental drawing.

nodes and edges. Here in order to see what the invisible nodes are, they are displayed in very light grey.

If we use this approach on the example in figure 4.1, we get the incremental drawing of the snapshots shown in figure 4.8. This represents a significant visual improvement in the relation between frames from our individual approach.

There is a small issue with this solution. Nodes representing the same data location are intended to be identical in each snapshot. Since we encode some temporary information such as the tree/DAG/cycle analysis result in node shapes, we do in fact end up changing the graph during its execution. That is because a rectangular node does not take as much space

as an elliptic node or as an hexagonal node. Therefore, the graph might move slightly up or down if tree/DAG/cycle is actually represented. In practice this problem is only mildly distracting, and it is obvious that figure 4.8 gives a much better result than we were able to achieve with tools we investigated in Section 4.1.1.

4.2 Numerical Summary

Although an acceptable snapshot representation and incremental layout is important. Many programs also produce very large data structures, whether or not they are modified frequently. Even a simple program such as BiSort from the JOlden benchmark suite generates more than 120,000 objects—far too many for a drawing tool to handle, or to meaningfully show on a screen or in an animation. Interactive visualization techniques can improve this situation, but it is clear that animations, and even representative snapshots are simply not feasible in all situations. For the benchmarks we analyze in the subsequent section we have thus concentrated on alternative representations that draw only reduced, aggregate information on data structure properties, and not the data structures themselves. We do so by giving a numerical summary of analysis information in the form of graphs.

From the analyses performed by the *J Shape Analyzer, we can get a number of graphs for each benchmark to better understand it and describe its properties. Those graphs are described below.

Tree/DAG/cycle

The first graph we can get is the number of entry points where the reachable nodes represent a tree, a DAG or a cyclic graph. This kind of graph is very useful to understand the evolution of data structures. For example, if we have a graph such as figure 4.9, we can see that nodes that form trees are perhaps being converted to DAGs.

GC info

The second kind of graph shows the number of live objects versus the number of dead objects not yet collected by GC. With this kind of graph, we can see that a program only

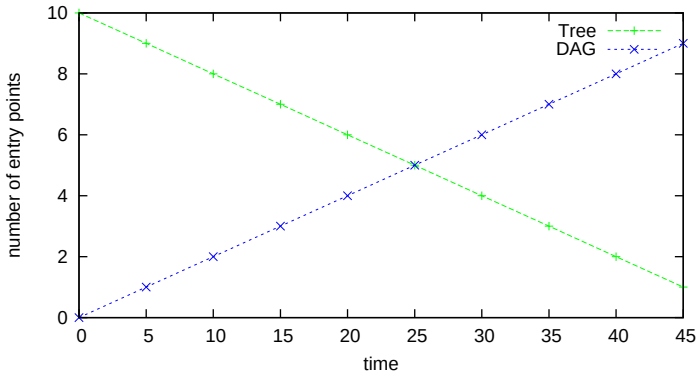


Figure 4.9: Example of a graph showing the number of entry point type. This graph shows that trees are converted to DAGs over time.

creates objects at the beginning and never deletes anything if we get a graph such as figure 4.10. However, if we have a graph where both the live object line and the GC line fluctuate a lot over the program execution period, we can conclude that GC was very busy during the execution of the program, and that objects were constantly created and thrown away. This kind of graph is very useful to understand the behaviour of the GC.

Connectivity

The third kind of graph shows the number of connected data structures. This kind of graph is useful in determining the maximum number of connected data structures and how it changes over time, and to see if there is any correlation between the number of entry points and the number of connected data structures.

Purity of entry points

The fourth kind of graph shows the number of pure versus the number of impure entry points. An entry point is considered to be pure if it follows the path *tree* –> *DAG* –> *cycle* without ever going backward, and it is considered impure otherwise as described in section 3.2.4. This kind of graph is useful in determining whether a static analysis will be useful or not, given optimal alias information.

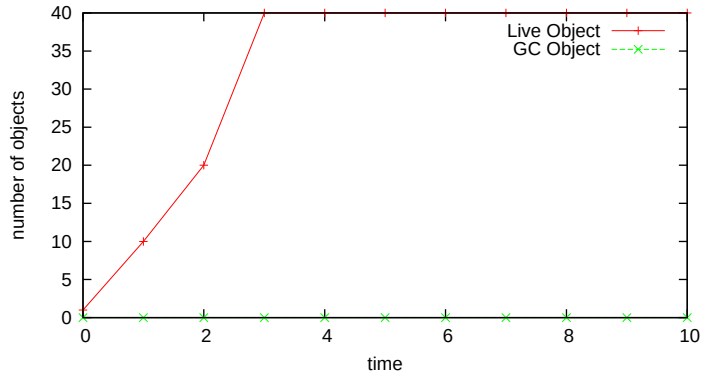


Figure 4.10: Example of a GC graph showing the number of live and dead objects over time. There are no dead, GC-able objects in this graph. Objects are created at the beginning and they are used throughout the whole program without adding more or deleting any.

Purity of types

The fifth and final kind of graph shows the number of distinct data types, where objects with the same static signature constitute a data type. The graph also shows whether the data types are pure or impure according to the definition of purity given in section 3.2.4. This graph is useful in determining the number of different data types present in the program. Similar to the fourth kind of graph, it is also useful in determining whether static analysis will be useful in analyzing the program, given poor alias information.

Timelines

In addition to the different kind of graphs described above, each of those graphs comes in two kinds of timeline. We can represent time by the number of updates done on data structures, or by the bytecode execution count. Each of those graphs may convey different information about the behaviour of the program. Take for example figure 4.11, the top graph shows a graph by the number of updates and bottom one shows a graph by bytecode count. By looking at the straight horizontal line in the middle of each of the graphs, the top one shows that there were many updates done on the data structures. We would naturally think that it took a lot of time to execute. However by looking at the bottom graph, it shows that in fact it took a really short time to execute. Using both forms of time axis, we can get

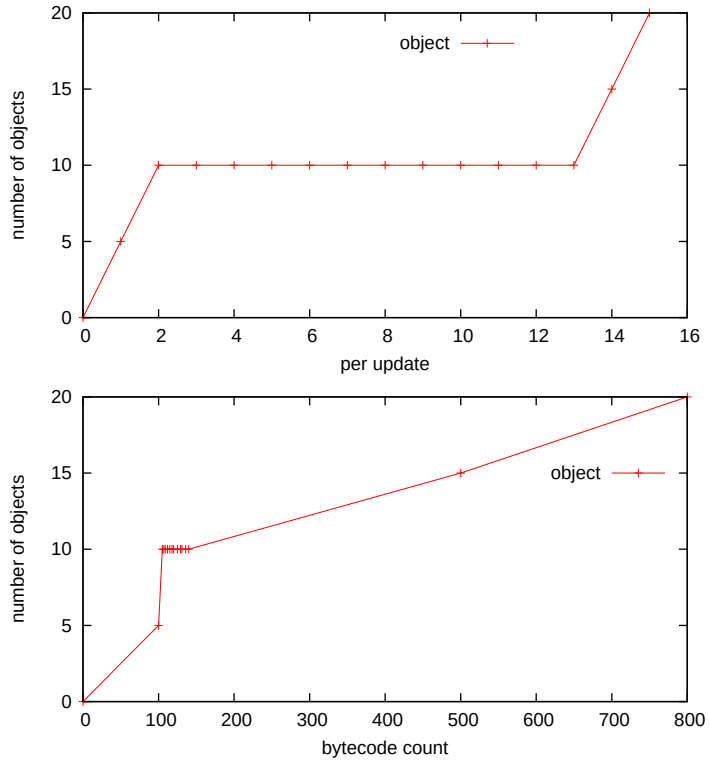


Figure 4.11: The top graph shows a graph by the number of updates, and the bottom one shows a graph over bytecode executed.

a different perspectives on the program execution and its behaviour.

Chapter 5
Experiments

We have analyzed a number of benchmarks from the SPECjvm98 [SPE98] and JOlden [CM01b] benchmark suites. Below we describe the programs analyzed, and present a visualization example in terms of snapshots and analysis examples based on the various data gathered using our framework. These discussions demonstrate both the kind of data we can collect, and also how it relates to relevant program features and behaviour.

The next section describes the benchmarks we analyzed, then we give an example of an animation done on a splay tree. In section 5.3 we give some results from our test at the combinatorial topology analysis. Finally in section 5.4 we analyze benchmarks from the SPECjvm98 [SPE98] and JOlden [CM01b] suites, then give our overall thoughts.

5.1 Benchmarks

We have analyzed benchmarks from three basic categories. The first kind consist of tiny programs designed to test the framework, and which are also suitable for snapshot visualizations. We used two well-known algorithms, a splay tree implementation and a red-black tree implementation. Both programs construct a small tree and then delete some nodes; below we only present the SplayTree benchmark program.

More realistic, but still manageable small results are obtained by analyzing benchmarks from the JOlden suite. These are small but non-trivial programs that focus on use of dynamic data structures. Benchmarks analyzed include Barnes-Hut, BiSort, Em3d, Power,

and TSP (Travelling Salesman Problem).

Our final category is of moderately large programs, taken from the SPECjvm98 suite, which have a more complex heap usage. The benchmarks analyzed here are Jess, MpegAudio, Compress, Javac, and DB. Jess is the only benchmark from the SPECjvm98 suite to run at full size (size 100). Compress and MpegAudio run at size 10, and the remaining benchmarks run at size 1. All benchmarks are run in Sun's 1.4.0 JVM, server mode (128M heap).

5.2 Snapshot Example

If a program is relatively small, and in general does not contain more than approximately 1,000 objects, a meaningful visualization of data structures updates can be produced where a snapshot is generated for each update. We use the *dot* tool in *GraphViz* [GN00] to layout the graphs, encoding node properties as discussed in sections 3.2.3 and 3.2.4.

In figure 5.1 and figure 5.2 we show incremental snapshots generated for the complete series of data structure updates performed in the SplayTree program. From a) to p) the splay tree is constructed by adding three tree nodes, and q) to x) shows the deletion of two tree nodes where those nodes are shown as garbage objects in dotted lines. As seen, these kinds of incremental snapshot of the data structure evolution over the program execution may be quite useful for understanding data structure operations and behaviour.

5.3 Combinatorial Topology Results

We already described the combinatorial topology analysis in section 3.2.4. Since this is only a test in using combinatorial topology on data structures, we will run the analysis on small examples. In this section we give and discuss results of the analysis done on a binary tree and on a grid.

Let us first look at the binary tree example shown in figure 5.3 (a). To use the combinatorial topology analysis, we have to assume that all edges have an edge that goes the other direction, which makes the graph fully doubly-linked. The equations we get from the

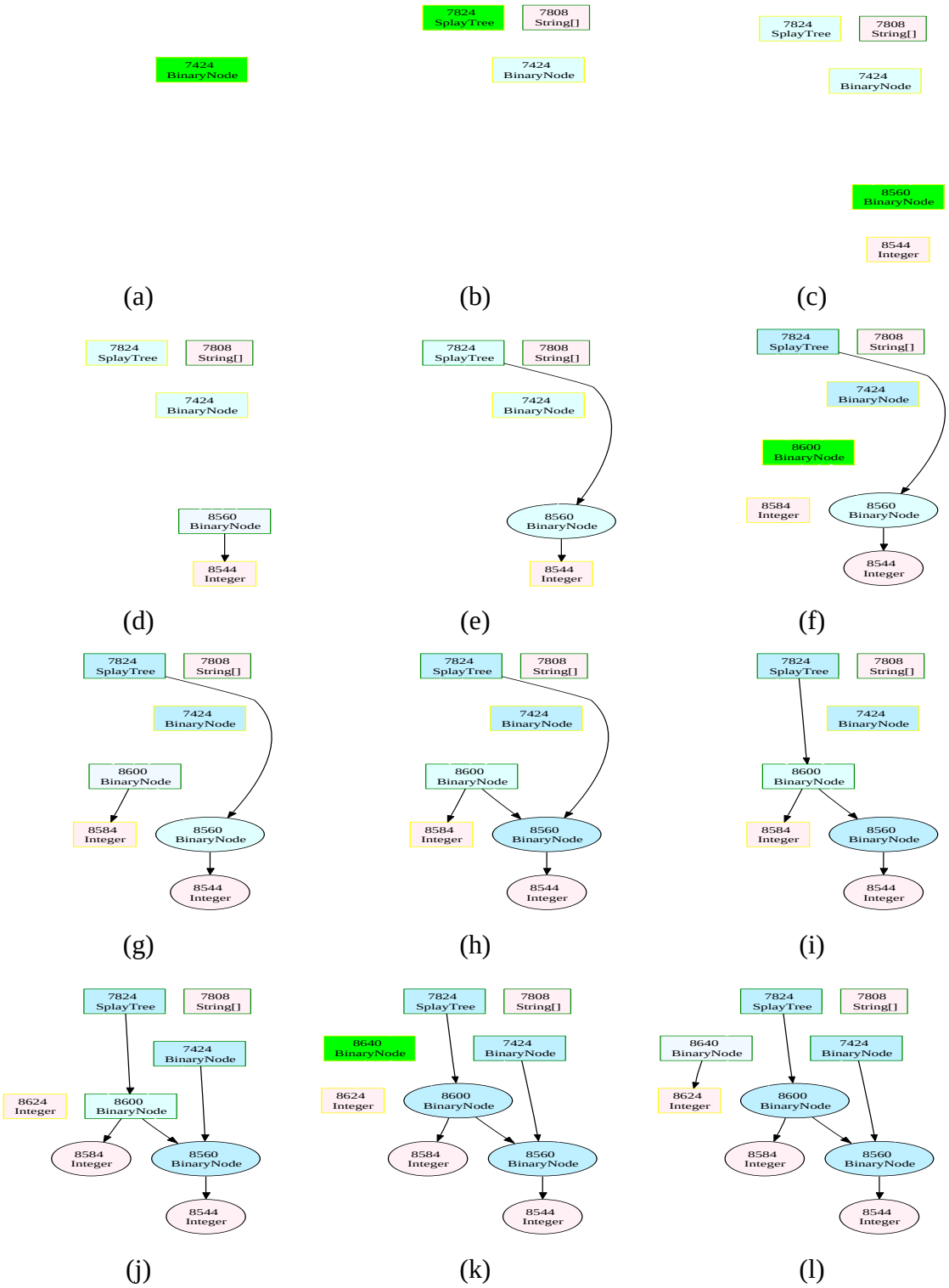


Figure 5.1: SplayTree snapshots (part 1).

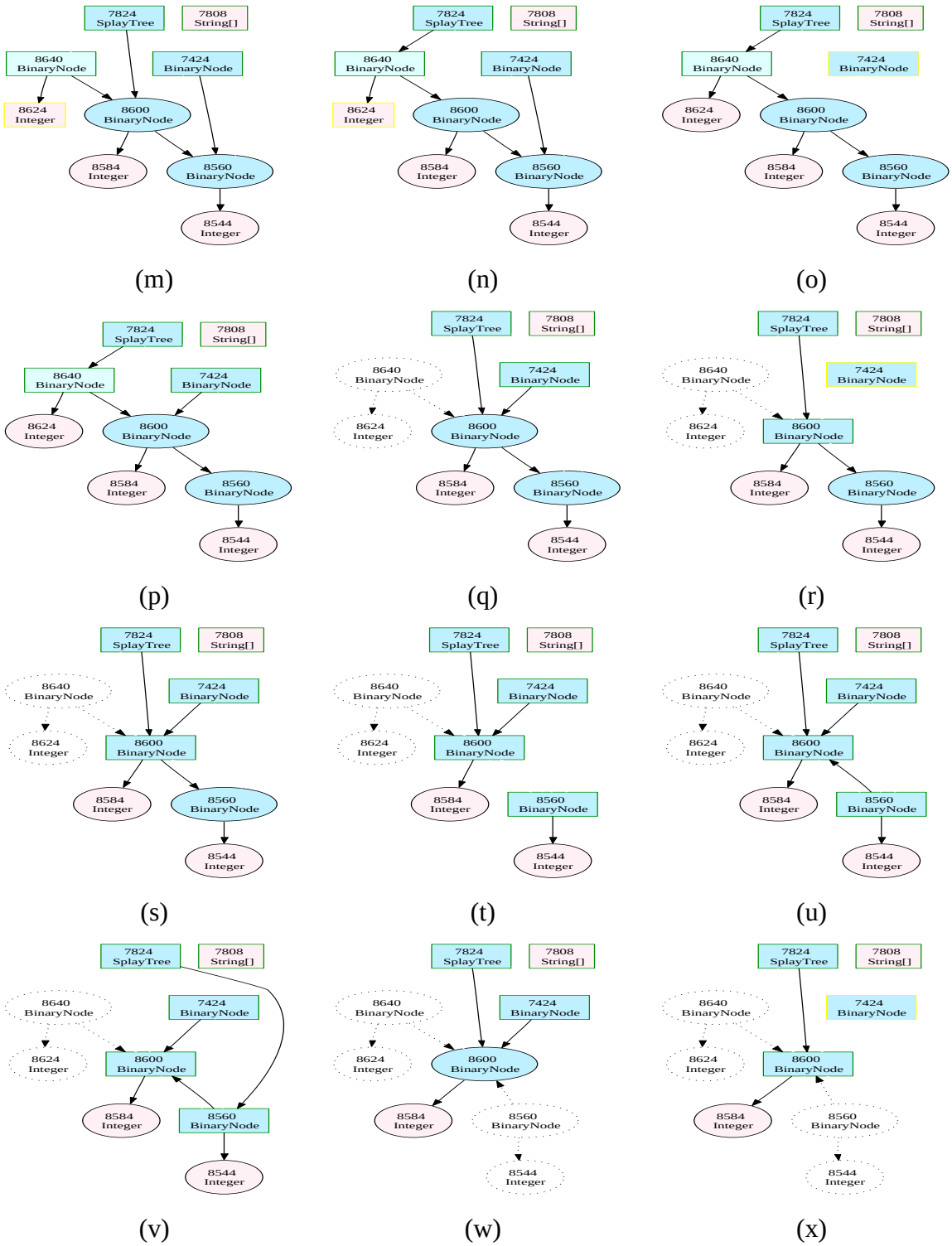
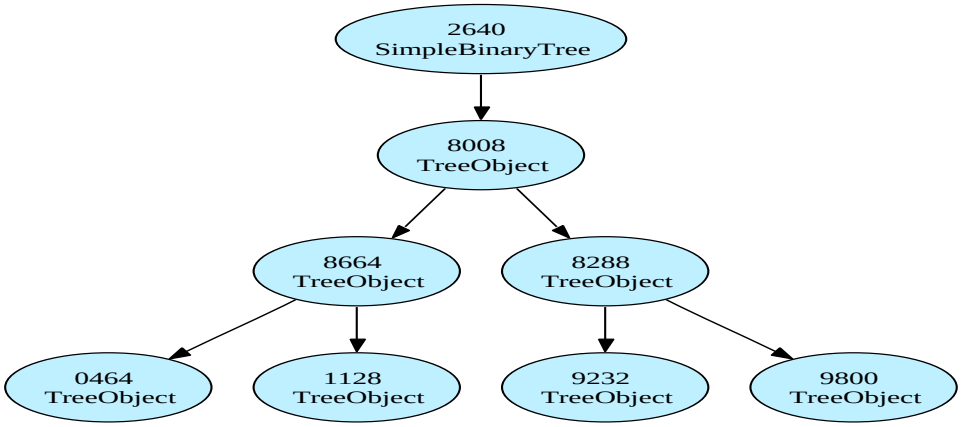
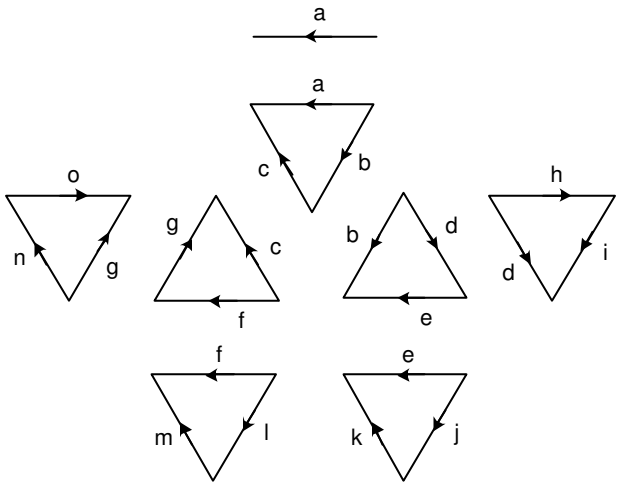


Figure 5.2: SplayTree snapshots (part 2).



(a)



(b)

Figure 5.3: (a) Shows a snapshot of a binary tree, and (b) shows the corresponding surface in combinatorial topology.

corresponding surfaces shown in figure 5.3 (b) are:

$$\begin{aligned} a^{-1}bc &= 1 & \qquad e^{-1}jk &= 1 \\ b^{-1}de &= 1 & \qquad a &= 1 \\ c^{-1}fg &= 1 & \qquad f^{-1}lm &= 1 \\ d^{-1}hi &= 1 & \qquad g^{-1}no &= 1 \end{aligned}$$

Inputing these equations in the combinatorial topology analyzer produces the following output:

```
Equation: -abc -bde -cfg -dhi -ejk a -flm -gno

Joining -abc and -bde to get c-ade
Joining c-ade and -cfg to get -adefg
Joining -adefg and -dhi to get efg-ahi
Joining efg-ahi and -ejk to get fg-ahijk
Joining fg-ahijk and a to get hijkfg
Joining hijkfg and -flm to get ghijklm
Joining ghijklm and -gno to get hijklmno
Input surface: hijklmno

Result: hijklmno
Result: hijklmno
Extra cuff.
Result:

handles=0, crosscaps=0, cuffs=1
B = 0
```

Listing 5.1: Output generated from the combinatorial topology analyzer given the above equations

From the output above, once the original equations are reduced, we have this resulting equation:

$$\text{Resulting Equation:} \qquad hijklmno = 1$$

We also get these parameter results:

handles = 0, crosscaps = 0, cuffs = 1

B = 0

The result shows that this snapshot of a binary tree is a surface with one cuff, which cannot be cut without dividing it into separate pieces. Note that although we only show a simple snapshot of a binary tree, the result we get from all the snapshots is the same as here. Results from analysis of a simple splay tree also give a similar result as shown here.

These results tell us, analytically, that a doubly connected tree structure maintains reachability between all nodes. For identifying actual shape, this technique is less useful. Consider the grid example shown in figure 5.4 (a). As in the binary tree example, we have to assume that all edges have an edge that goes the other direction to make the graph fully doubly-connected. The corresponding surface shown in figure 5.4 (b) gives this set of equations:

$$\begin{aligned} a &= 1 & h^{-1}e^{-1}jk &= 1 \\ a^{-1}bc &= 1 & j^{-1}g^{-1}lm &= 1 \\ b^{-1}de &= 1 & i^{-1}no &= 1 \\ d^{-1}fg &= 1 & n^{-1}k^{-1}pq &= 1 \\ c^{-1}hi &= 1 & p^{-1}m^{-1}rs &= 1 \end{aligned}$$

Once these equations are reduced, we have this resulting equation:

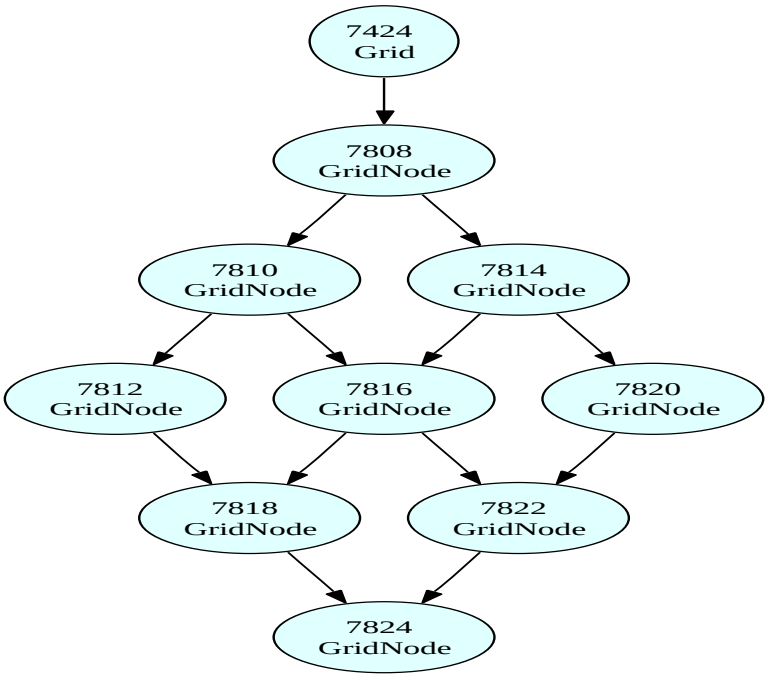
$$\text{Resulting Equation:} \qquad qoflrs = 1$$

We also get these parameter results:

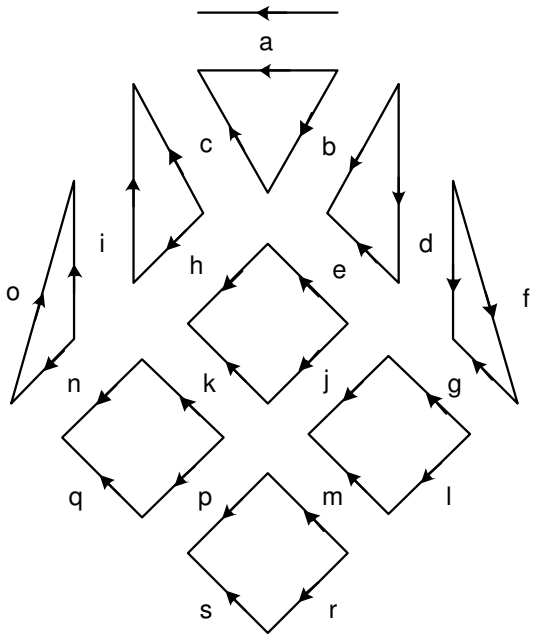
handles = 0, crosscaps = 0, cuffs = 1

B = 0

Both structures thus reduce to a surface with one cuff. This is interesting with respect to determining reachability or partitionability of data structures for parallelization, but forms too coarse a categorization of shape for more general data structure analyses. Thus while this technique helps demonstrate the ability of our framework and approach to accumulate new analyses, further investigation and application of this analysis are left for future work.



(a)



(b)

Figure 5.4: (a) Shows a snapshot of a grid, and (b) shows the corresponding surface in combinatorial topology.

5.4 Analysis & Numerical Summary Results

Non-trivial benchmarks are not amenable to literal data structure representations, and so we present aggregated data from analyses run in the shape analyzer at each data structure modification. We use data from three main analyses: a tree/DAG/cycle shape classification, reachability analysis, and purity.

Shape classification data is based on the number of entry points that reach single-node, tree, DAG, and cycle type data structures, plotted over time. For portability of results, time is measured abstractly, as either bytecodes executed, or in terms of number of data structure modifications. To compress the visual representation, data shown is also sometimes a sampled subset; sample periods vary up to every 100k updates, and are indicated in the individual descriptions.

Reachability is given both in terms of the number of live versus dead objects, and in terms of number of connected structures. The former makes it easy to see general trends in volume of data and garbage, and for a limited visual inspection of GC drag. The latter gives a better impression of the number of connected data structures (of size a least 2) actually used in the program.

Purity data is used in two forms, entry point purity and type-based purity, as described in section 3.2.4.

Furthermore, using the analyses mentioned above, we can also detect program phases. This feature would be quite useful for dynamically adaptable systems as they need to accurately detect changes within procedures.

The goal of this experiment is to see how much information we can gather about the programs' behaviour on data structures using all the analyses already mentioned. An observation for each benchmarks investigated is described in the following sections where we work in parallel with the source code to confirm our observations.

Only benchmarks which extensively make use of the heap were chosen from each benchmark suites. However, a few of them, especially from the SpecJVM98 suite, were left out as they took too much time to be analyzed. We have to note that Jess from the SpecJVM98 suite, which is one of the more complex benchmark, took more than two weeks to be analyzed.

5.4.1 JOlden Suite

This section shows analysis results for BiSort, Barnes-Hut, Em3d, Power, and TSP along with some analysis of the graphs. The complete set of graphs for each benchmark can be found following the link provided in appendix A.

BiSort

BiSort performs two bitonic sorts, one forward and one backward. It works in two phases. The first phase is the tree construction, and the second phase is the sorting. Our analysis is done on BiSort sorting 128k integers.

In figure 5.5 we can easily see the first phase, where the tree is being constructed. A number of single nodes are allocated, and then consumed by construction of the base tree. At about 1/3 of the way through execution the program enters its second phase; here many changes are performed on the tree, and the number of tree structures becomes quite variable. As the tree is modified the data types fluctuate between DAG types and tree types in a complementary fashion: nodes are being rearranged, and not copied or deleted. Note that there are not in fact as many disjoint structures as the number of trees and DAGs would indicate; call chains and recursive calls in particular allow for the stack to contain multiple entry points to the same structure, magnifying the apparent number of structures. This is more evident in figure 5.6 — in the second phase there is only ever 1 or 2 connected structures.

Figure 5.7 gives an indication of how well a static analysis could do in identifying the data structure shape, assuming perfect alias information. Most references are pure, but the second phase of execution contains several impure variables due to the tree modifications. Statically these references would have to be considered DAGs. Less optimal alias information may spread this conservative approximation. Furthermore it is not surprising to see that figure 5.8 shows some impurity in the fields since we have impure entry points.

Figure 5.9 reinforces the observed phase behaviour of the data structures: objects are allocated (tree construction), followed by a long period of relative stability. Interestingly, there are no dead objects, an observation compatible with our claim that the data structure is modified by moving nodes, not adding or deleting.

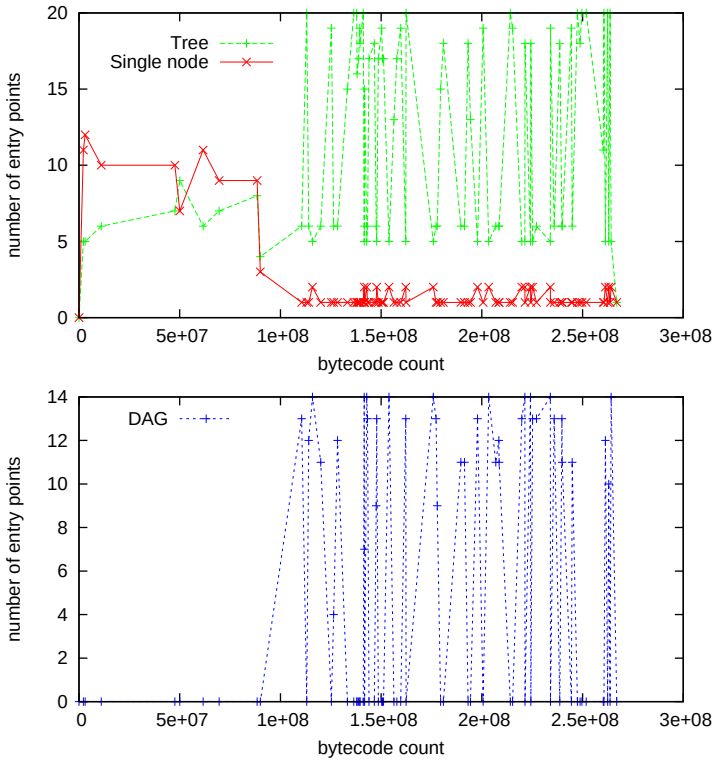


Figure 5.5: BiSort analysis results by bytecode for every 10k updates. The top figure shows single nodes and trees over bytecodes executed, and the bottom figure shows DAGs. There are no cycles in BiSort.

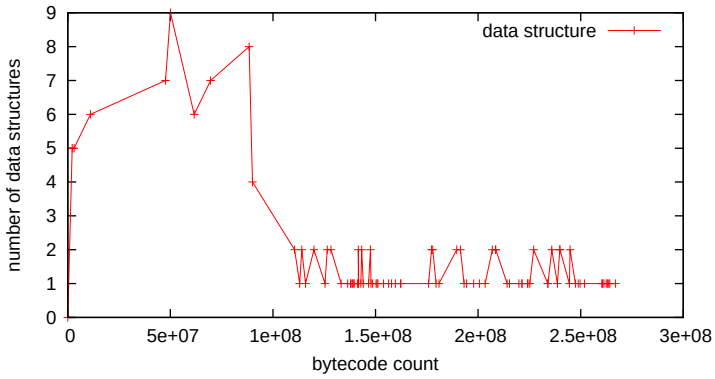


Figure 5.6: BiSort analysis over bytecode executed showing the number of connected data structures for every 10k updates.

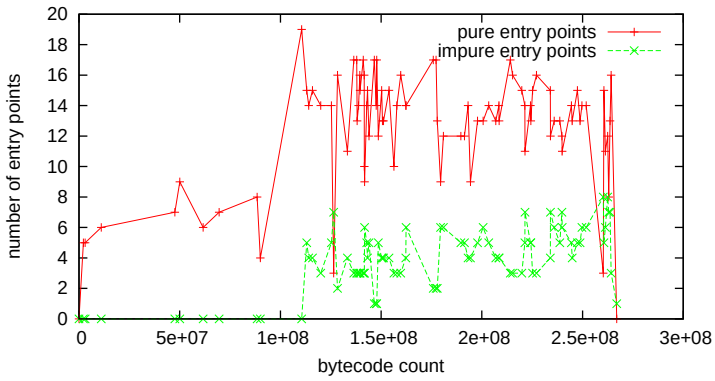


Figure 5.7: BiSort analysis over bytecode executed showing the number of pure vs. impure entry points for every 10k updates.

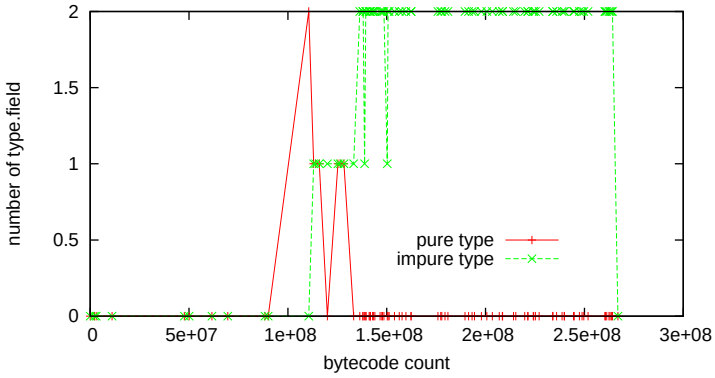


Figure 5.8: BiSort analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every 10k updates.

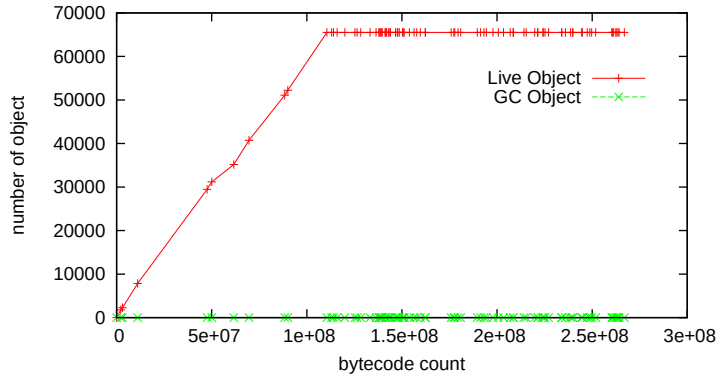


Figure 5.9: BiSort GC results by bytecode for every 10k updates, showing the number of live and dead objects over bytecodes executed. There are no dead objects in Bisort.

Barnes-Hut

Barnes-Hut solves the classic N-body gravitational attraction problem. Barnes-Hut works in two phases; first is the tree construction, where a quad-tree is constructed, and second is the force computation, where the tree is traversed. Our analysis of Barnes-Hut is done with 2000 bodies.

From the graph in figure 5.10 it is evident that this program is quite dynamic in behaviour, and aggressive and frequent GC is used to limit the amount of accumulated garbage. As with BiSort there are no cyclic data structures at all. This is unsurprising for tree-based programs, but is also informative: it suggests, for instance, that the quadtree does not make use of parent pointers in child nodes.

The phases are not obvious in the shape information, but are clearly shown in the GC results graph of figure 5.11. The large spikes in number of dead objects indicate a rapid accumulation of garbage data, and the short-lived nature of the spikes suggests this is temporary data, quickly collected. The frequent variation in number of tree and DAG entry points is in this case mainly due to the use of allocated, intermediate data. Purity data shown in figure 5.12 and in figure 5.13 show all references are pure, further supporting the conclusion that the shapes of existing data structures are not generally altered.

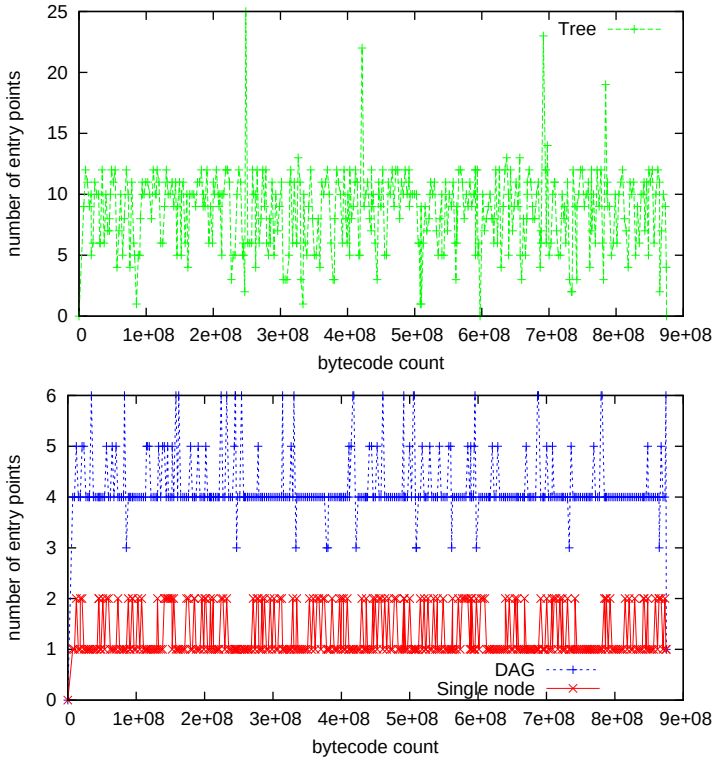


Figure 5.10: Barnes-Hut analysis results by bytecode for every 1k updates. On the top figure is shown the number of single node and tree entry points over "time" (bytecodes executed), and on the bottom the number of DAGs. Again, there are no cyclic structures.

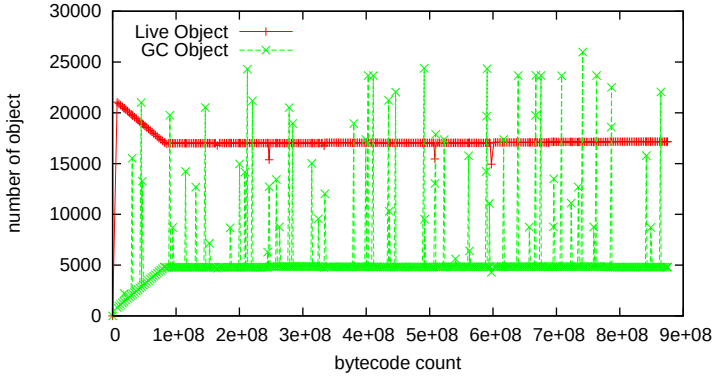


Figure 5.11: Barnes-Hut GC results by bytecode for every 1k updates, showing the number of live and dead objects over bytecodes executed.

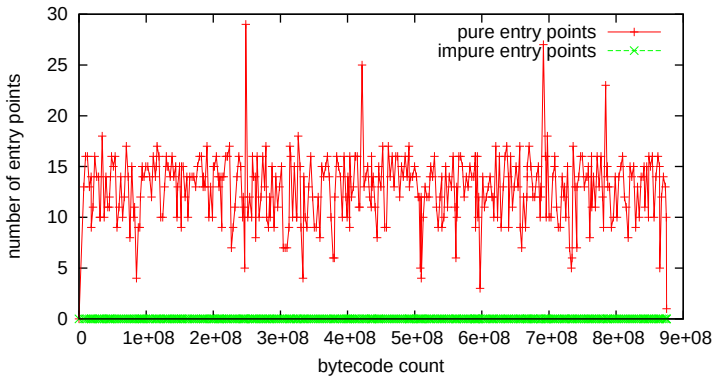


Figure 5.12: Barnes-Hut analysis over bytecode executed showing the number of pure vs. impure entry points for every 1k updates. There are no impure entry points in Barnes-Hut.

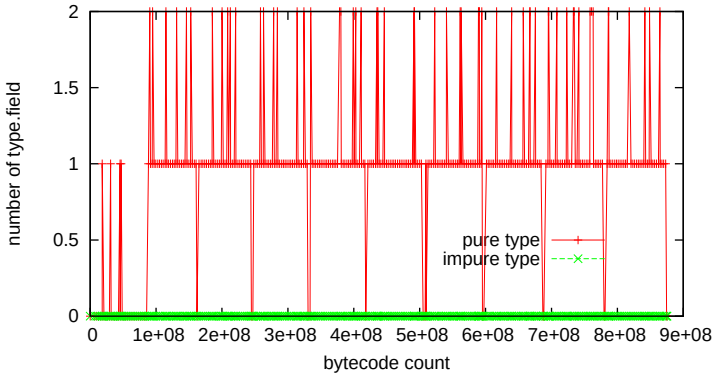


Figure 5.13: Barnes-Hut analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every 1k updates. There are no impure types in Barnes-Hut.

Em3d

Em3d simulates the propagation of electro-magnetic waves through 3D object using nodes in an irregular bipartite graph to represent electric and magnetic field values. For our analysis, Em3d simulated 2000 nodes of out-degree 100.

In figure 5.14, we can see that during the total execution of the program there are at most 5 trees and 1 DAG at any point. Data structures in Em3D are quite few as shown in figure 5.16, and the ratio of live nodes to entry points suggests a limited number of larger data structures are used. In fact, there is mainly a table of linked lists.

Behaviour is relatively stable throughout this benchmark, at least until near the end of the program. At that point the data structures are reduced to a couple of single nodes and one tree. In this case we are able to see the effect of tearing down the data structures, something much less evident in the previous benchmarks. The conversion of data to garbage at the end of the program is confirmed by figure 5.15, where garbage rises as live objects reduce in number.

The stable behaviour can also be seen in figure 5.17 and in figure 5.18, where the purity data shown in terms of both entry points and type indicate that all references are pure; again the shapes of existing data structures are not generally altered.

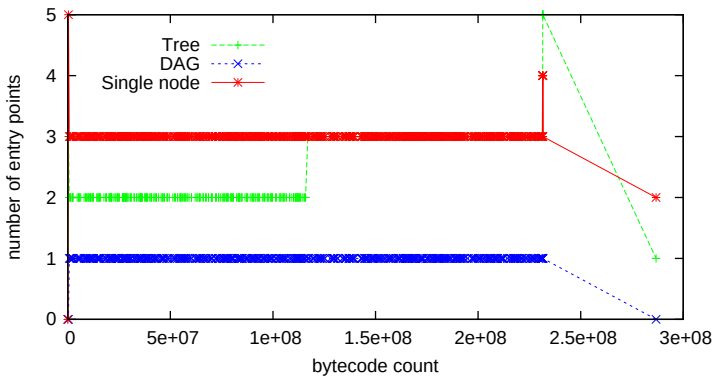


Figure 5.14: Em3d analysis result by bytecode for every 1k updates. Single nodes, trees, and DAGs are shown in this figure. There are no cycles in Em3d.

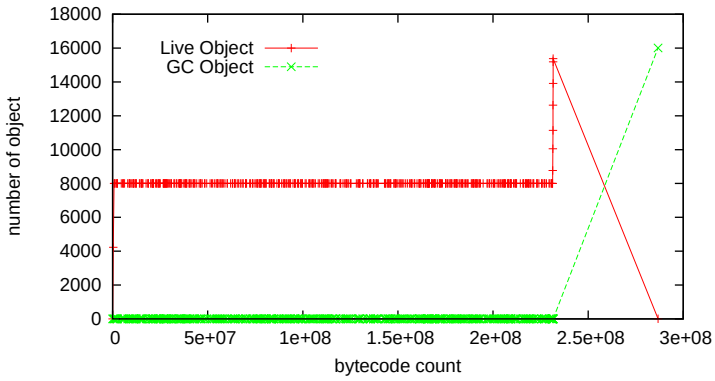


Figure 5.15: Em3d GC result for every 1k updates, showing the number of live and dead objects over bytecodes executed.

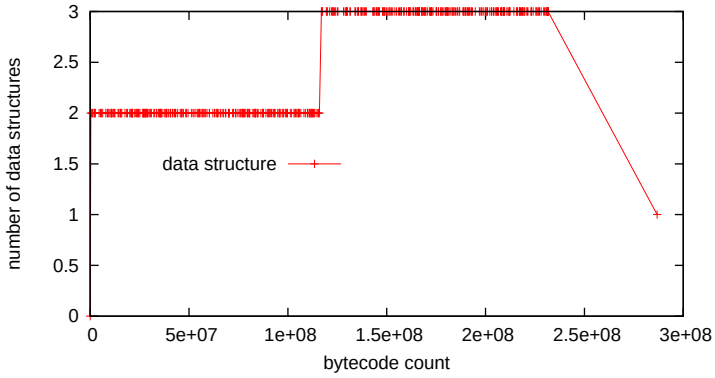


Figure 5.16: Em3d analysis over bytecode executed showing the number of connected data strucures for every 1k updates.

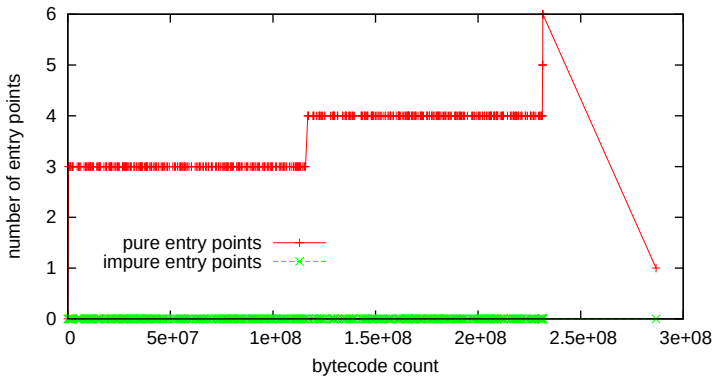


Figure 5.17: Em3d analysis over bytecode executed showing the number of pure vs. impure entry points for every 1k updates. There are no impure entry points in Em3d.

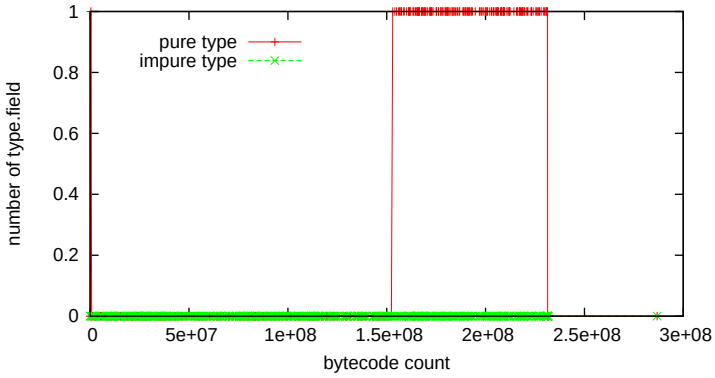


Figure 5.18: Em3d analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every 1k updates. There are no impure types in Em3d.

Power

Power solves the Power System Optimization Problem, where the price of each customer's power consumption is set so that the economic efficiency of the whole community is maximized. It works in two phases. The first phase is the tree construction, and the second phase is the price computation. The analysis of Power is done on 10k customers.

Figure 5.19 shows there are only trees and single nodes present. This is consistent with the algorithm as it only constructs trees. Once the tree construction is completed, the behaviour is fairly stable as seen in figure 5.21 and the bottom graph of figure 5.20, where the number of live and dead objects, and the number of data structures remain fairly stable.

Although the second phase seems to be fairly stable in terms of object and data structure creation, we can see from figure 5.19 that it is not as the number of entry points fluctuates a lot. This means that although there is not much data structure activities, the program is by no mean idle. It is infact using and accessing the created data structures as seen by the fluctuation in the number of entry points.

From the top graphs of figure 5.19 and figure 5.20, we can see that the tree construction phase occurs within a very short time frame. However, from the top graphs, we can see that it consists of roughly half of the total data structure changes. This difference is more showing in section 5.4.2 when we describe results from Compress.

Figure 5.22 and figure 5.23 show that although all entry points are pure, we do not have any purity result for the types. That is due to the fact that all the entry points are objects from `main`; therefore they have no parents nodes. To better explain this take for example figure 5.24 (a), where `ObjectB` is the entry point and it is the left child of `ObjectA`. For that example, `ObjectB`'s data type is `ObjectA.left`. However in (b) where `ObjectA` is the entry point, since it does not have a parent node, we do not have a type. We know it is a variable from main, but it is pointless to keep track of those since we cannot merge objects having the same static signature as all variables in `main` are unique.

Travelling Salesman Problem

TSP computes an estimate of the best Hamiltonian circuit for the Travelling Salesman Problem. There are two clear phases evident in both figures 5.25 and figure 5.28; a short

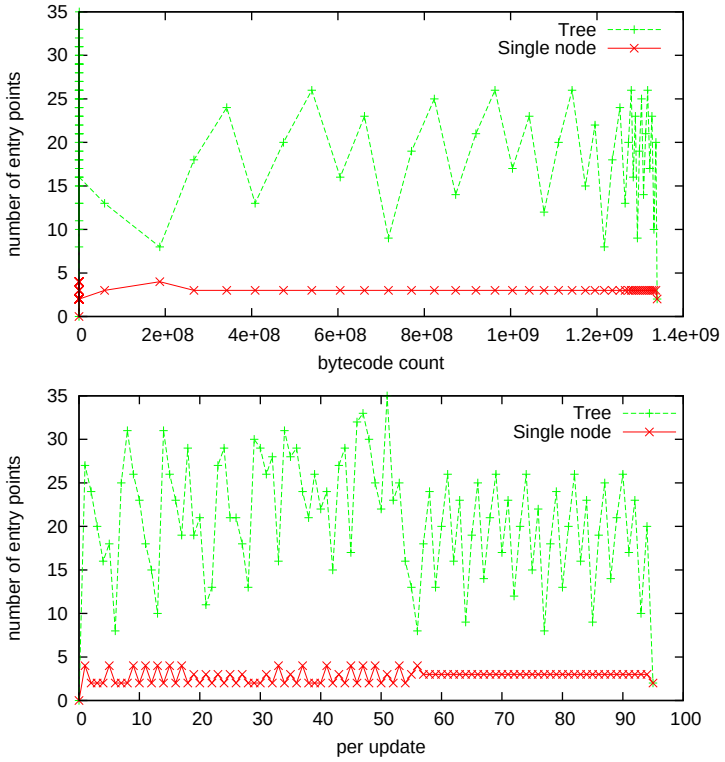


Figure 5.19: Power analysis result for every 1k updates. The top graph is plotted with respect to the total bytecodes executed, and the bottom graph with respect to the total number of data structure changes. Both graphs show the number of single nodes and trees. There are no DAGs or cycles in Power.

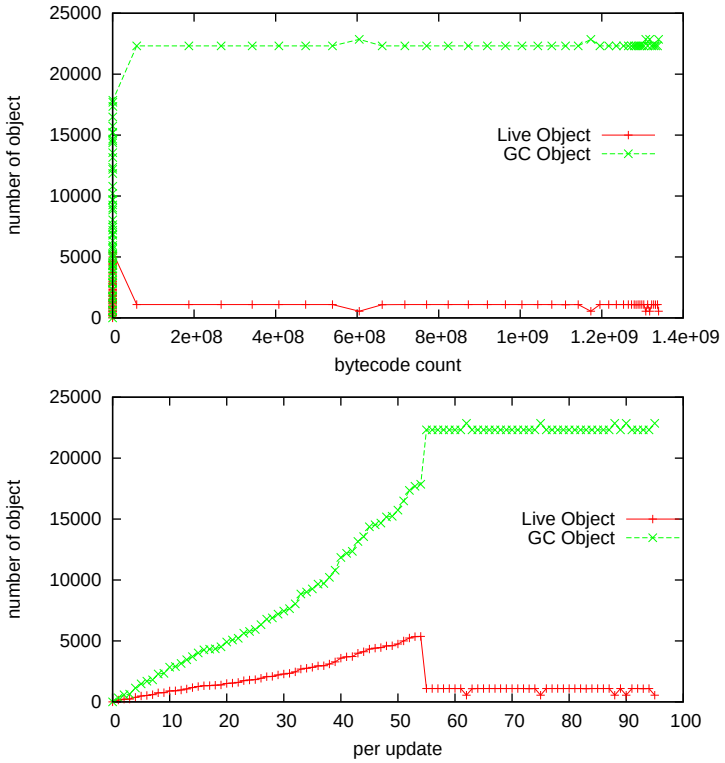


Figure 5.20: Power GC result for every 1k updates. At the top the time axis is in terms of bytecodes executed, and at the bottom in terms of total data structure updates.

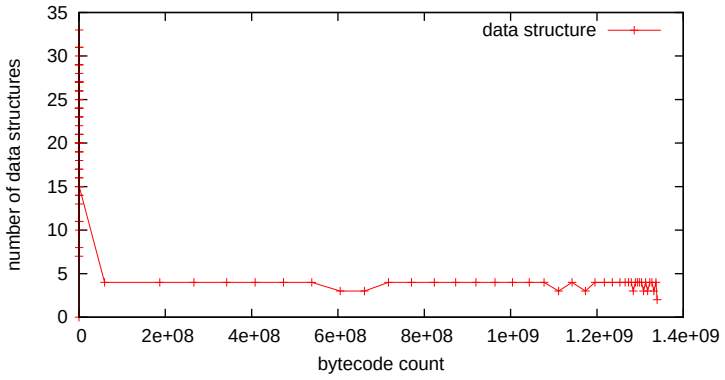


Figure 5.21: Power analysis over bytecode executed showing the number of connected data structures for every 1k updates.

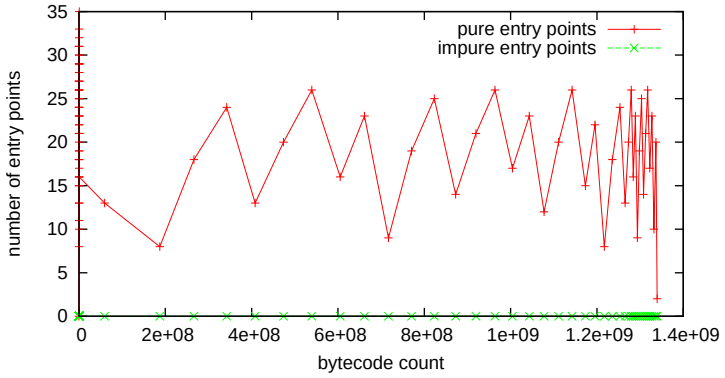


Figure 5.22: Power analysis over bytecode executed showing the number of pure vs. impure entry points for every 1k updates. There are no impure entry points in Power.

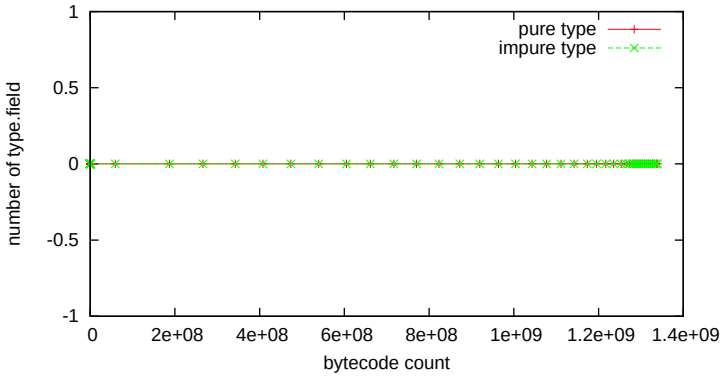


Figure 5.23: Power analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every 1k updates. There are neither pure nor impure type results in Power.

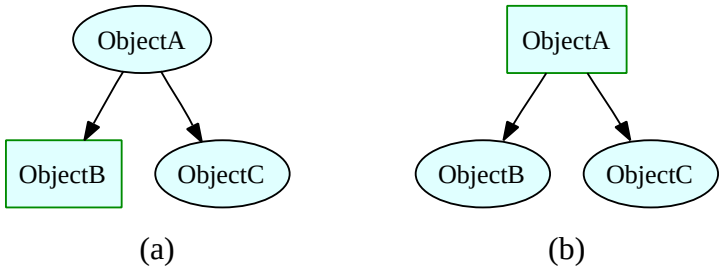


Figure 5.24: Tree example with `ObjectA` as the root, `ObjectB` as the left child and `ObjectC` as the right child. In the left graph, `ObjectB` is the entry point, and in the right graph the entry point is `ObjectA`.

initial phase constructing the problem, and a longer phase of analysis. The analysis of TSP is done on 10k cities.

TSP is our first presented benchmark to actually include cyclic data structures. There are also a very large number of tree data structures, orders of magnitude more than single nodes, DAGS, or cycles. In fact the algorithm mainly builds trees, and the few cycles can be attributed to a double-linked threading of trees forming partial solutions to the input problem.

References are uniformly pure as shown in figure 5.26 and figure 5.27. This suggests a mainly static heap structure. However, since the number of entry points in different shape categories does fluctuate, the data structures clearly do change. In this program the use of heap data at different stages in the computation is well-separated—entry points used in processing and generating the tree structures are disjoint from those used for DAGs and for cyclic structures.

There is no garbage collection apparent in figure 5.28. However, the number of live objects decreases dramatically twice; there is necessarily some garbage generated by these reductions. In this benchmark the generation of dead nodes and their collection occurs between snapshots, leaving no direct evidence of dead nodes in our sampled results. Larger, more detailed graphs or actual numbers would reveal this difference. In terms of general trends, though, it is clear that TSP, particularly in comparison with Barnes-Hut, does not produce or carry much garbage.

5.4.2 SPECjvm98 Suite

This section shows analysis results for Jess, Compress, MpegAudio, DB, and Javac along with some analysis of the graphs. The complete set of graphs for each benchmark can be found following the link provided in appendix A.

Jess

Jess produces a lot of structures of all types, although most of them are single node objects, as shown in figure 5.29. There are no cycles, and there is a rhythmic pattern of tree/DAG construction. This behaviour roughly corresponds with the algorithm and input, which does

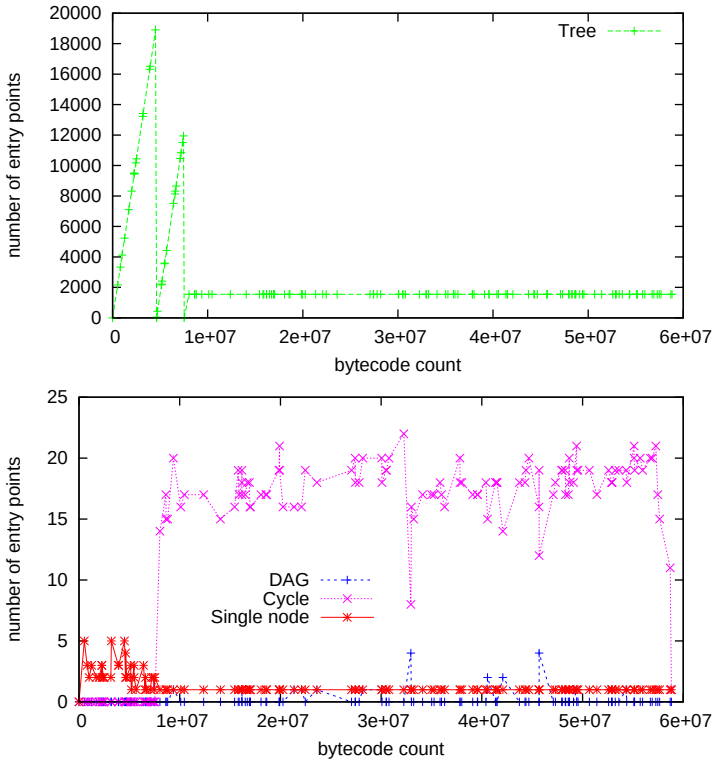


Figure 5.25: TSP analysis results by bytecode for every 1k updates. On the top are trees, and on the bottom single nodes, DAGs and cycles.

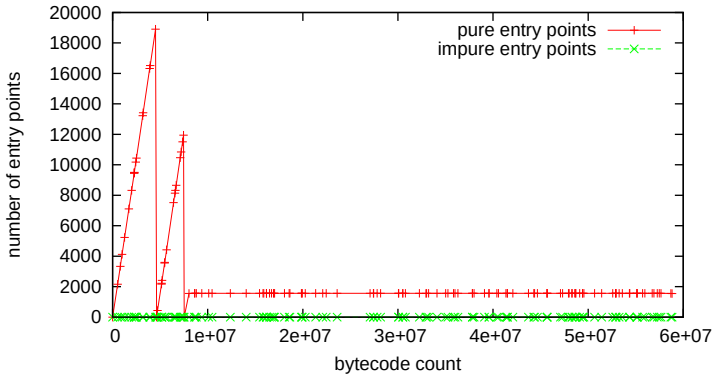


Figure 5.26: TSP analysis over bytecode executed showing the number of connected data structures for every 1k updates. There are no impure entry points in TSP.

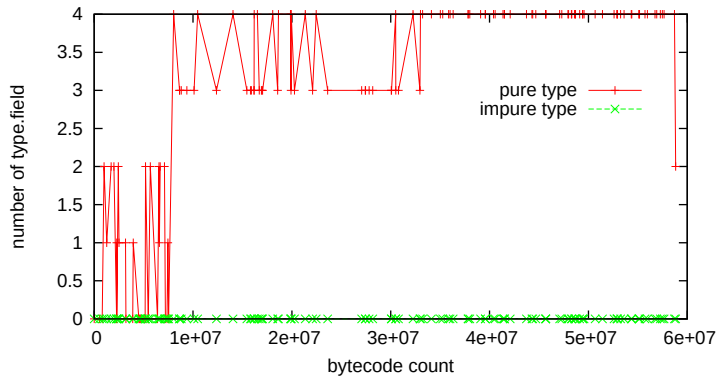


Figure 5.27: TSP analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every 1k updates. There are no impure types in TSP.

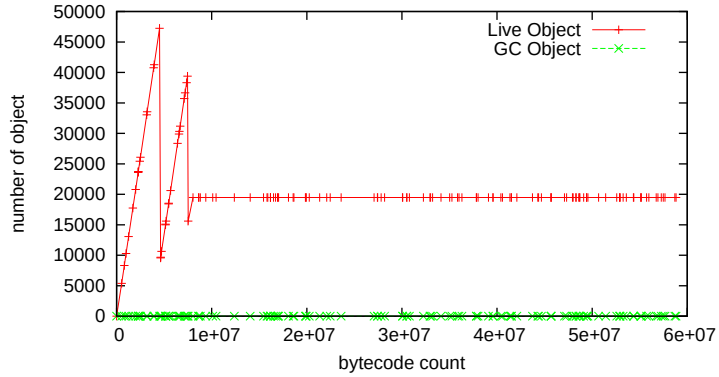


Figure 5.28: TSP GC results by bytecode for every 1k updates. Again, there are no dead objects evident in this graph.

repeated, tree-based searches to solve an input combinatorial problem. This behaviour can be better seen in figure 5.30.

Memory usage in Jess is more complicated than in the JOlden programs. From figure 5.31 we can see that a large number of objects are dead, usually many more than are live at any one time. Moreover, while the live set is overall stable, the number of dead nodes seems to have a a general upward slant, increasing over time. This is also true of single node structures shown in figure 5.29.

We believe this to be an artifact of heap adaptation. Jess allocates a lot of temporary objects (single nodes). The heap pressure due to the use of temporary object allocations results in the heap being expanded to accommodate the perceived memory requirements. However, the core, necessary and retained data is not increasing, and a larger heap merely provides more room for garbage to accumulate. In this situation the amount of drag increases as the heap increases, suggesting that more aggressive GC rather than increasing heap size may result in more efficient execution.

Compress

Most of the benchmarks produce extremely similar graphs whether the time axis is formed of bytecode executions, or expressed in terms of data structure modifications, where data structure updates are quite regular. Compress shows this is not always the case. In the bottom graph of figure 5.32 the number of entry points are plotted against total number of data structure updates. The results shows quite regular behaviour, with three obvious phases of execution, each consisting of two sub-phases. This correlates nicely with the known behaviour of Compress under our input parameters, which is to compress and decompress three files. The three phases in Compress are also evident in pattern formed by garbage objects shown in figure 5.33.

The top graph shows the same data plotted with respect to bytecodes executed. Here the phases are considerably less evident—the time spent compressing and decompressing each file is clearly uneven. Regularity of changes is a useful property for adaptive program optimization. Compress is quite deterministic in the sequence of action executed, but duration of program phases, a large part of predicting behaviour, is here an input property.

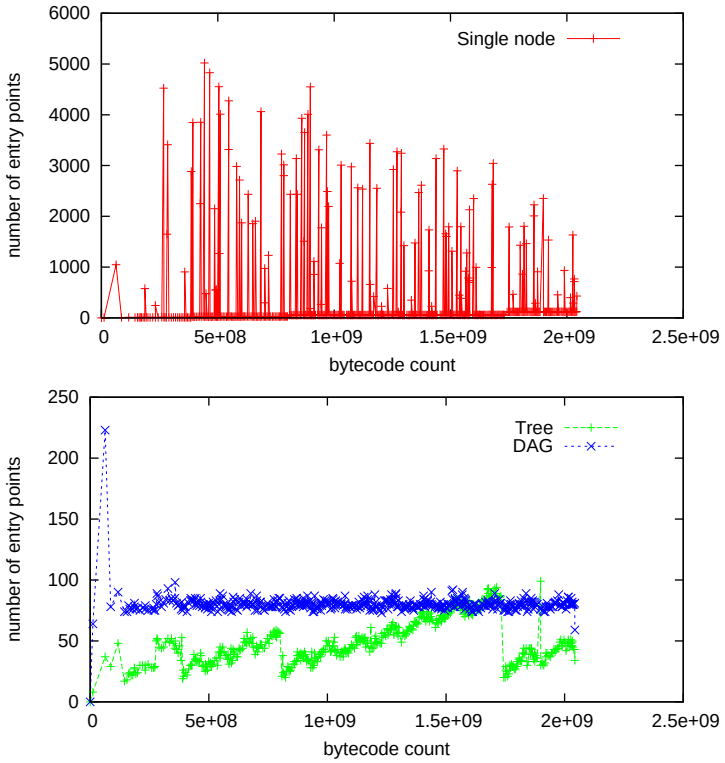


Figure 5.29: Jess analysis results by bytecode for every 100k updates. On the top are single nodes, and on the bottom trees and DAGS. There are no cycles in Jess.

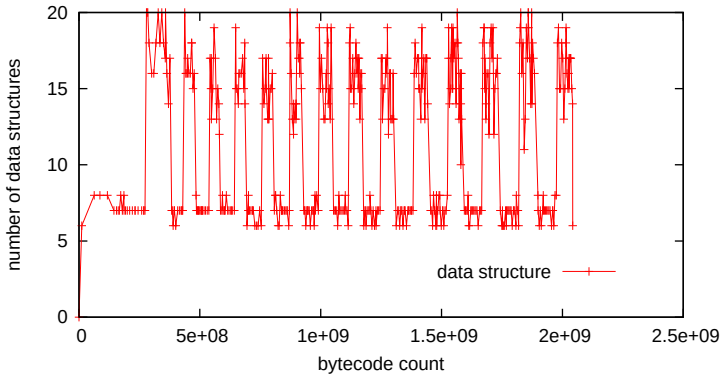


Figure 5.30: Jess analysis over bytecode executed showing the number of connected data structures for every 100k updates.

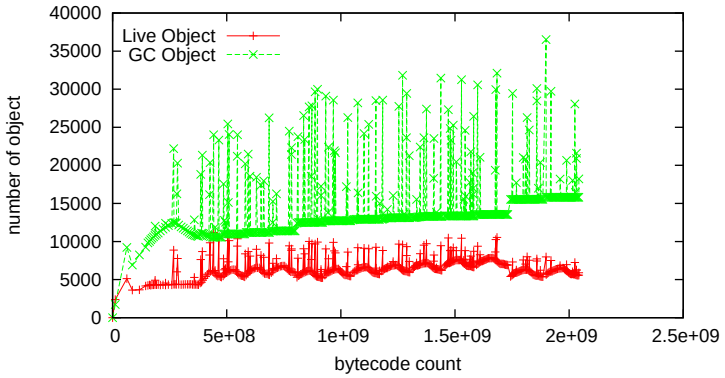


Figure 5.31: Jess GC results by bytecode for every 100k updates.

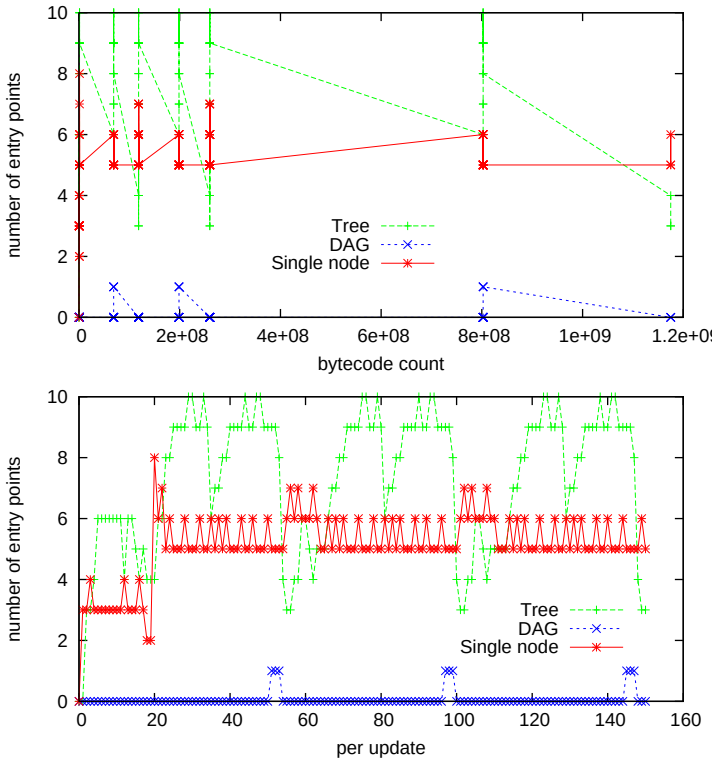


Figure 5.32: Compress shape analysis result by both bytecodes executed (above) and number of heap updates (below) for every update. There are no cycles in Compress.

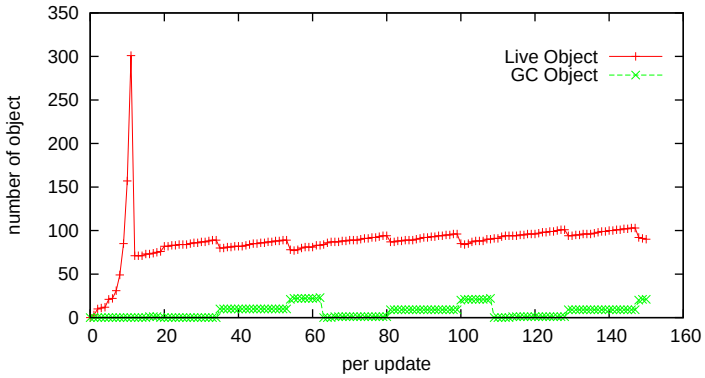


Figure 5.33: Compress GC result for every update, showing the number of live and dead objects in terms of total data structure updates.

MpegAudio

In this benchmark, we will show graphs in terms of total number of data structure updates instead of bytecode executed as they show more information.

MpegAudio decodes a compressed audio file twice under our input parameters, and figure 5.34 clearly shows the two phases. Moreover, MpegAudio demonstrates the potentially large effect of good alias analysis on a static shape analysis. Figure 5.35 shows that while there are a large number of entry points, they are entirely pure. However, a shape analysis that relies on less precise alias information may not be as successful as this suggests—figure 5.36 shows that when minimal alias data is available there are proportionally many impure reference.

DB

DB performs several database functions on a database stored in memory. Therefore, the first phase of this program is to construct the database, and the second phase is to retrieve data from it.

The top graph of figure 5.37 and figure 5.38 show that the first phase takes the major part of program execution, about two thirds of the time just constructing the database.

Figure 5.39 and figure 5.40 show that all references are pure. This is an unsurprising result for a program that only utilizes trees. Since only trees and single nodes are used in

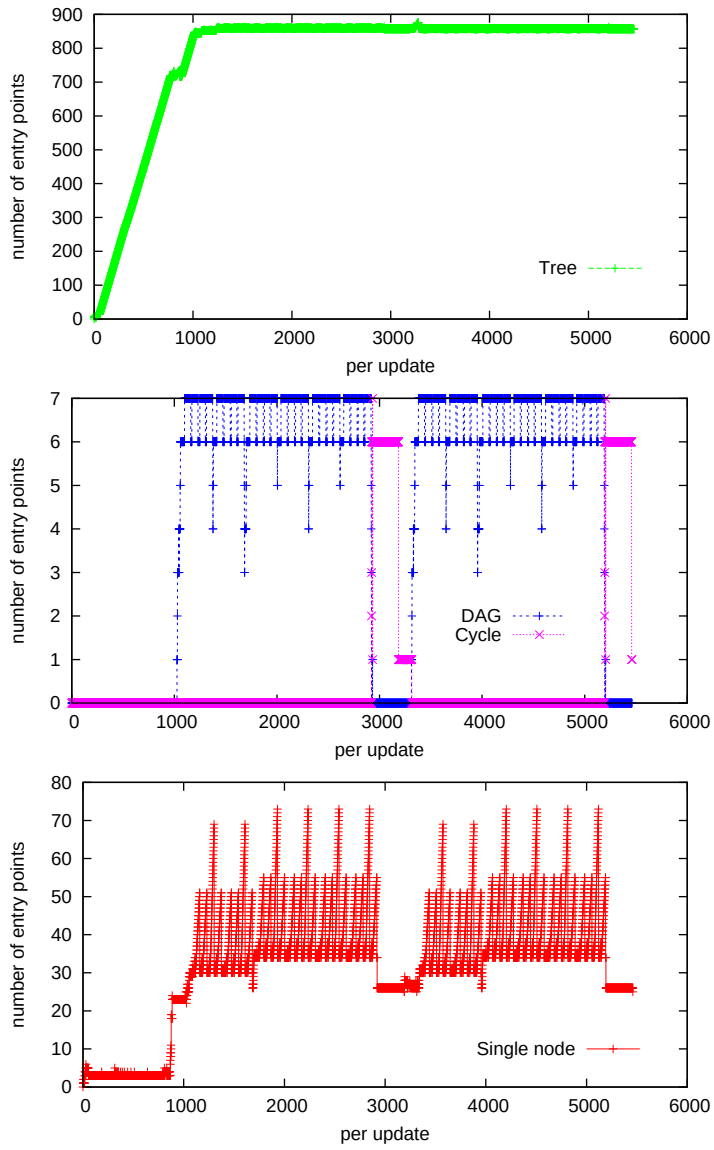


Figure 5.34: MpegAudio analysis result for every updates with respect to the total number of data structure changes, where the top graph shows the number of trees, the middle shows the number of cycles and DAGs, and the bottom one shows the number of single nodes.

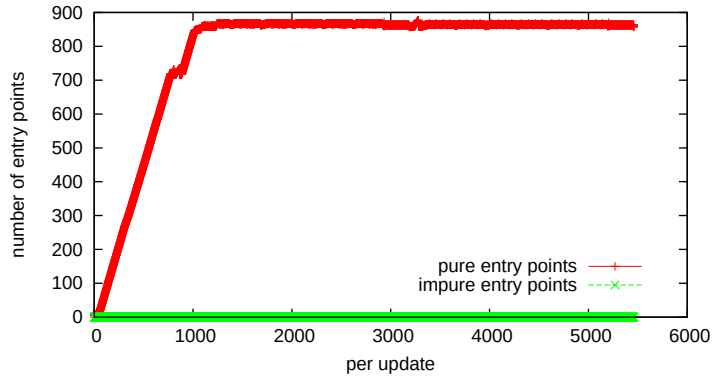


Figure 5.35: MpegAudio analysis for every updates with respect to the total number of data structure changes, showing the number of pure vs. impure entry points. There are no impure entry points in MpegAudio.

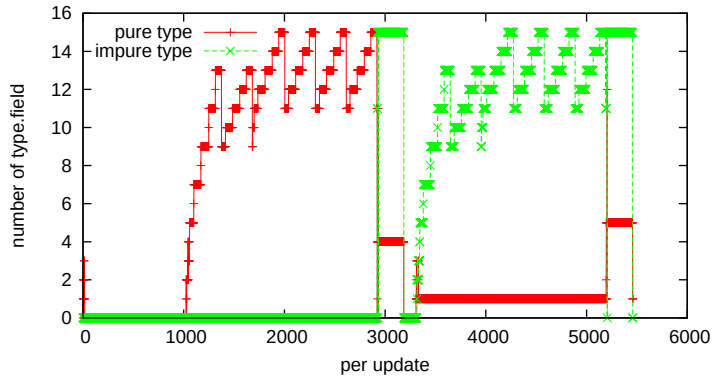


Figure 5.36: MpegAudio analysis for every updates with respect to the total number of data structure changes, showing the purity result of fields merged over all objects of the same class type.

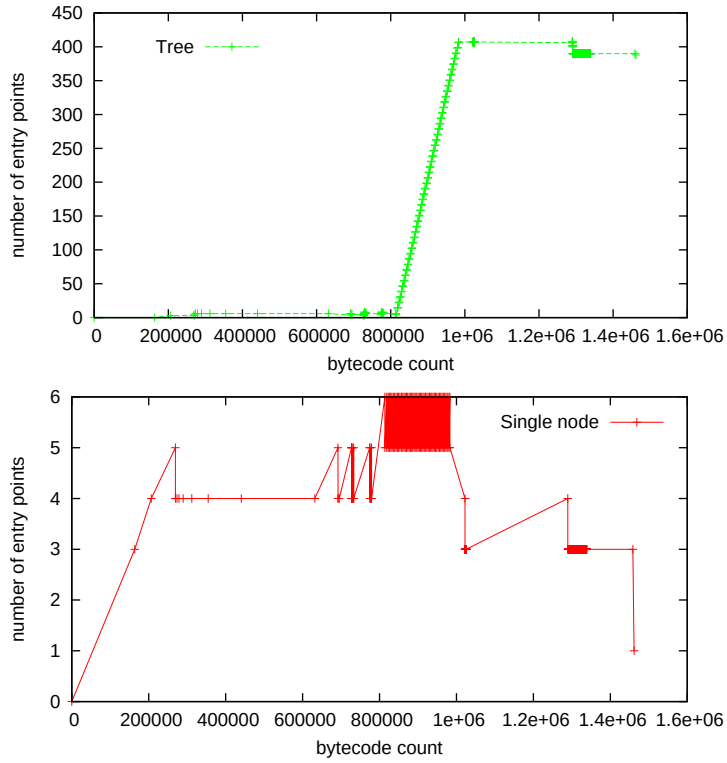


Figure 5.37: Db analysis results over bytecode executed for every update. On the top are trees, and on the bottom single nodes. There are no DAGs or cycles in Db.

this program, there is no way for an entry point or a type to ever be impure. Nevertheless it demonstrates how simple the use of data structure can be even in a non-trivial program.

Javac

Javac is the JDK 1.0.2 Java compiler compiling JavaLex. From figure 5.41 and figure 5.43 it is obvious that there are two major phases, where the first phase does not produce much objects, and the second phase consists of 2 sub-phases. This is consistent with its behaviour since the first phase is to scan the file to ensure the correct grammar, and this does not require significant object allocations. The second phase begins by creating the abstract syntax tree, and this is evident by the increase in objects, trees and connected data structures, and also in the reachability results of figure 5.42. Symbol tables creation is the

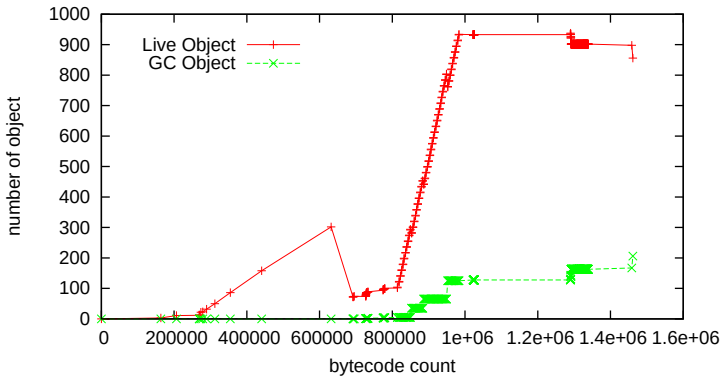


Figure 5.38: Db analysis over bytecode executed showing the number of live and dead objects for every update.

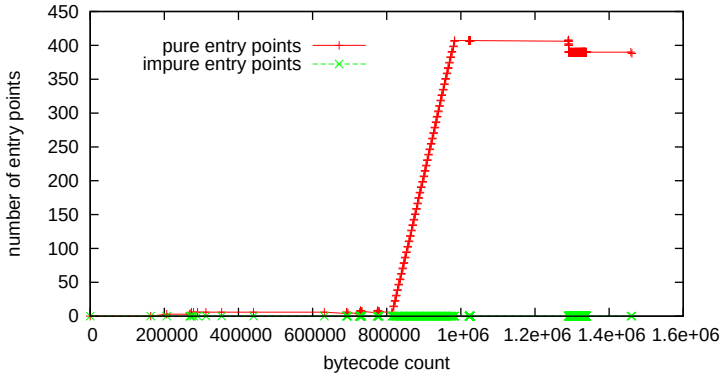


Figure 5.39: Db analysis over bytecode executed showing the number of pure vs. impure entry points for every update. There are no impure entry points in Db.

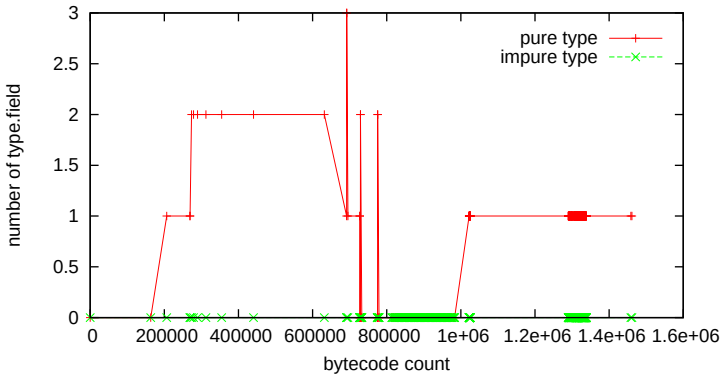


Figure 5.40: Db analysis over bytecode executed showing the purity of fields merged over all objects of the same class type for every update. There are no impure types in Db.

second part of that phase, and this can be seen in the number of trees (top of figure 5.41) and in the number of connected data structures shown in figure 5.43.

5.4.3 Summary

Even small Java programs make extensive use of heap structures, and so a dynamic data structure analysis has the ability to provide a great deal of information about program execution. Literal snapshots of data structures are most informative, but do not in general scale to being able to represent real program data. Even from a simple tree/DAG/cycle descriptions of data structures, however, a surprising amount of detail on program behaviour is discernible in our numerical summary graphs. Most obviously execution phases are clearly visible in most of our graphs—programs, especially industry benchmarks, tend to behave in relatively regular ways, and data-centric algorithms show a corresponding regularity in data manipulations. Regularity is also seen in the kind of data used: the composition of trees, DAGs and cycles shows that while most of our benchmarks do perform numerous data structure modifications, they do not generally tend to be complex in their usage—there are surprisingly few cyclic structures found in our suites. In fact, there are surprisingly few actual connected structures (as opposed to entry points) in most programs. This lack of complexity in data structure usage is further supported by our purity data—certainly some entry points in some benchmarks do vary in the shape found, but most entry points are in

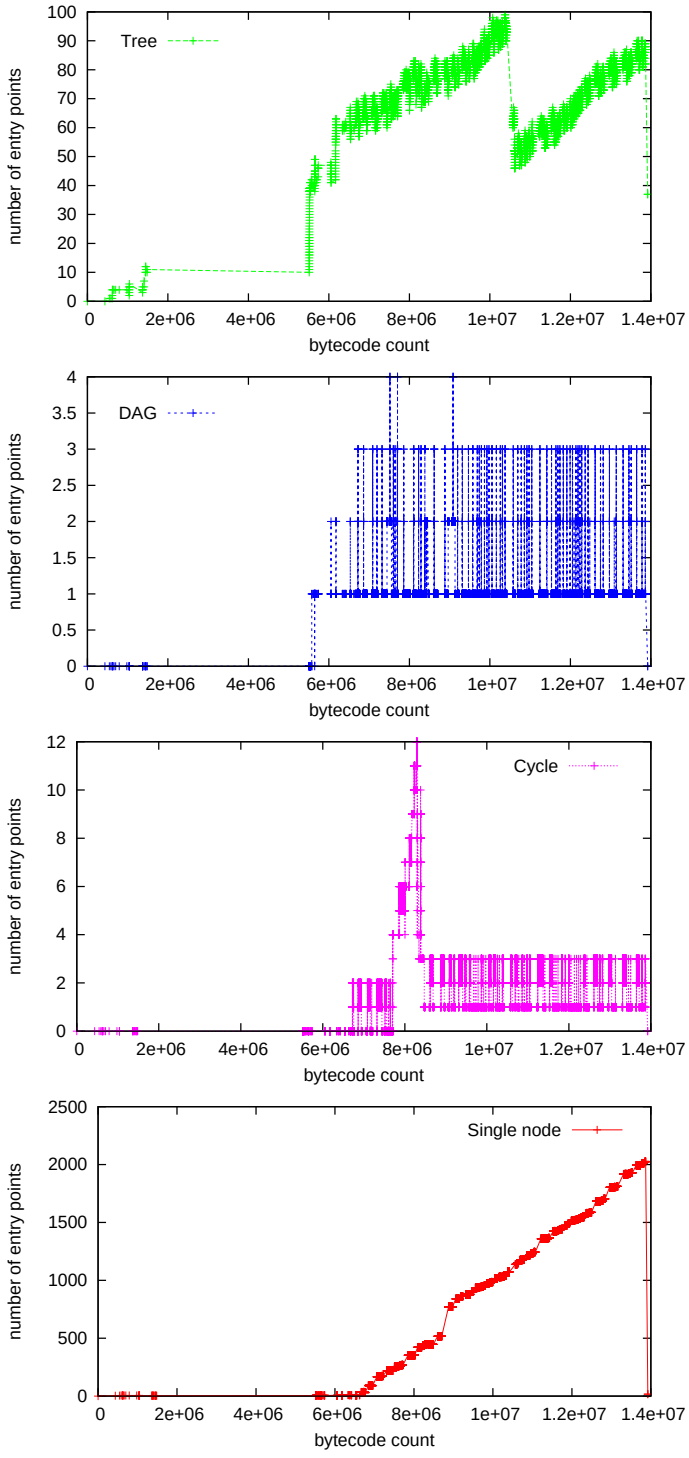


Figure 5.41: Javac analysis results over bytecode executed for every 10 updates. The graphs shown from top to bottom are trees, DAGs, cycles, and single nodes.

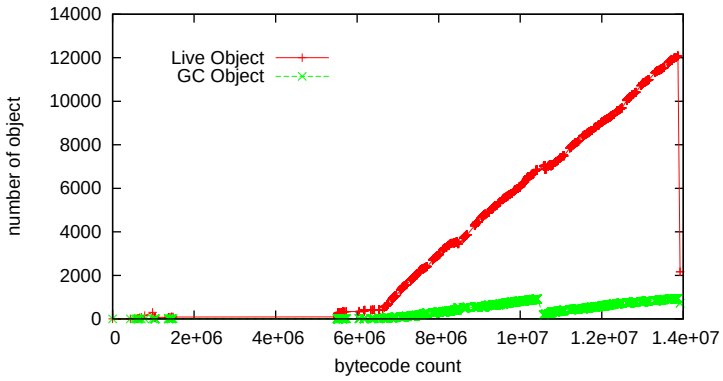


Figure 5.42: Javac GC results over bytecode executed for every 10 updates, showing the number of live and dead objects.

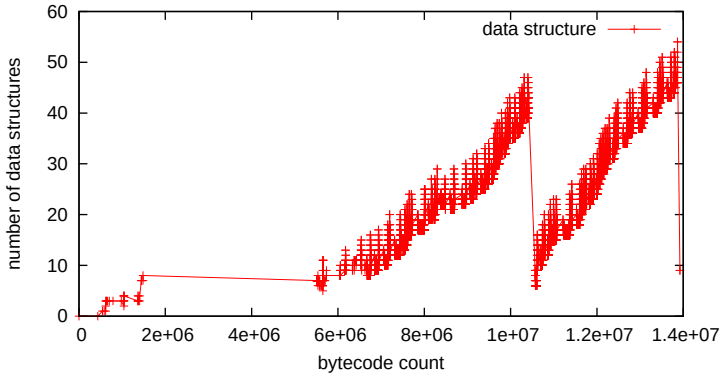


Figure 5.43: Javac analysis over bytecode executed showing the number of connected data structures for every 10 updates.

fact pure, with many of our programs even having 100% of entry points pure. This is less well reflected in type-based purity, indicating the importance of alias information, but is still quite encouraging for static approaches.

The impact of garbage on memory use is also intriguingly variable. Barnes-Hut and Jess generate great amounts of garbage, and certainly in the latter case dragged dead objects can be seen as a potentially important factor. Other benchmarks, such as TSP and BiSort carry little to no garbage, and may benefit from a corresponding reduction in GC; these benchmarks are not strongly GC-dependent.

Chapter 6
Conclusions and Future Work

6.1 Conclusions

This thesis has shown that dynamic data structure analysis has the ability to show detailed information on various aspects of program behaviour. This can help identify program characteristics, heap usage, and provide general understanding of any calculable static or evolving dynamic data structure property, advancing various optimization, understanding, and analysis goals.

By comparing our runtime data with that achievable through static means we have been able to verify that static approaches have potential to be quite accurate, at least for many of our example programs.

Our framework design and experience have demonstrated the feasibility of this technique, and also highlighted the research challenges involved. Extracting and reconstructing data structure changes is itself a non-trivial effort. Furthermore, we have demonstrated that our framework has all necessary information to support the implementation of various analyses such as tree/DAG/cycle, connectivity, purity, and combinatorial shape analysis.

We have shown two ways of representing the data gathered, divided into two main categories based on the size of the program. For small programs, literal representations are natural and provide maximal information. Even at this scale, however, the task is not trivial. Usable animations, as we have shown, is not as simple as one would think. It is

unfortunately also the case that this kind of representation does not scale very well to larger benchmarks.

More abstract analysis data in the form of numerical graphs is more suitable for larger benchmarks. This summarizes data structure analysis results over execution time. Although this representation does not give us the same kind of information as the literal one, we can still provide useful and interesting information on program behaviour, while maintaining much of the accuracy provided by a dynamic, runtime analysis.

6.2 Future Work

Dynamic data structure analysis is complex, and the pursuit of accurate, complete results implies a great many potential future directions for this work.

More benchmarks

More, and larger benchmark programs would of course be useful, as would an examination of benchmarks under different inputs. Our results here suggest data structure usage is often quite simple; further experimental evidence is needed, however, to make strong, general conclusions.

More analyses

We are also interested in evaluating the efficacy and accuracy of more detailed shape analysis techniques, such as those based on compact graph abstraction [NCA+04], or shape types [WSR00].

More efficient data gathering

Our analysis currently works offline, after the program is done executing. In order to be more efficient, we can run our analysis online by integrating our analysis using Aspect oriented programming [Asb02]. With the help of dynamic weaving, we will be able to analyze data structures during the program's run-time and only retrieve information that

we really require instead of reporting all events as it is currently done with *J. That way the analysis will not be as time consuming as it currently is. Efficiency improvements are important in order to scale dynamic data structure analysis to very larger programs. As described in section 5.4, it took more than two weeks to analyze Jess from the SpecJVM98 benchmark suite.

Mapping to source code

We have compared our dynamic data to that achievable by static approaches. This could be more refined by considering the state of variable at each static statement in the program. Our dynamic data, for instance, can be mapped to static code locations for direct comparison with static algorithms. This can help guide and measure static algorithm design, and would be a straightforward extension of our implementation.

Visualization improvements

Visualization improvements are many of course. We have been most recently working on improving animation quality by adapting existing tools to support incremental, if perhaps sub-optimal, graph drawing. Integration of good animation with interactive visualization techniques can help alleviate some of the scaling concerns with literal representations, and can also be the basis for useful educational and debugging tools. Further, novel visualizations that compactly summarize graph properties are also important, and a combined approach that allows inspection of both literal and more abstract representations of heap activity would greatly assist the understanding of how programs use data structures.

Appendix A
Complete Benchmarks Graphs

A.1 Benchmark Results

The complete set of graphs for the benchmarks we have covered from the JOlden and the SPECjvm98 suites can be found online following this link:

`http://www.sable.mcgill.ca/~spheng/graphs.html`

Bibliography

[Asb02] R. Dale Asberry. Aspect oriented programming (aop): Using aspectj to implement and enforce coding standards. `http://www.daleasberry.com/newsletters/200210/20021002.shtml`, 2002.

[BDE+02] Rhodes Brown, Karel Driesen, David Eng, Laurie Hendren, John Jorgensen, Clark Verbrugge, and Qin Wang. STEP: A framework for the efficient encoding of general trace data. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Anaylsis for Software Tools and Engineering (PASTE)*, New York, New York, United States, November 2002. ACM Press.

[BS01] Jeff Bogda and Ambuj Singh. Can a shape analysis work at run-time? In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[CAZ02] Francisco Corbera, Rafael Asenjo, and Emilio Zapata. New shape analysis and interprocedural techniques for automatic parallelization of C codes. *Int. J. Parallel Program.*, 30(1):37–63, 2002.

[CM01a] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java controller. In *PACT01*, pages 280–291, Barcelona, Spain, September 2001.

[CM01b] B. Cahoon and K.S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Barcelona, Spain, September 8-12 2001.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 149–168, 2003.

[DGK02] Stephan Diehl, Carsten Görg, and Andreas Kerren. Animating algorithms live and post mortem. In *Revised Lectures on Software Visualization, International Seminar*, pages 46–57, London, UK, 2002. Springer-Verlag.

[Duf04] Bruno Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, Montréal, Québec, Canada, 2004. URL: <`http://www.sable.mcgill.ca/metrics/`>.

[ECGN99] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[FM97] Pascal Fradet and Daniel Le Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996.

[GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
URL: <`citeseer.ist.psu.edu/gansner99open.html`>.

[GOP03] Thomas Gschwind, Johann Oberleitner, and Martin Pinzger. Using run-time data for program comprehension. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 245, Washington, DC, USA, 2003. IEEE Computer Society.

[HHN92] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, 1992.

[HLLF05] Abdelwahab Hamou-Lhadj, Timothy C. Lethbridge, and Lianjiang Fu. Seat: A usable trace analysis tool. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 157–160, Washington, DC, USA, 2005. IEEE Computer Society.

[HN90] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. In *IEEE Transaction on Parallel and Distributed Systems, Vol. 1, No. 1*, pages 35–47, January 1990.

[HR05] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 310–323, New York, NY, USA, 2005.

[Jam55] Robert C. James. Combinatorial topology of surfaces. *Mathematics Magazine*, 29:1–39, 1955.

[KS93] Nils Klarlund and Michael I. Schwartzbach. Graph types. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 196–205, New York, NY, USA, 1993.

[LD97] Mark Leone and R. Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report No.490, Computer Science Department, Indiana University, September 1997.

[LY96] Tim Lindholm and Frank Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1996.

[NCA+04] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E.L. Zapata. A new dependence test based on shape analysis for pointer-based codes. In *LCPC '04: Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[PJ02] Tony Printezis and Richard Jones. GCspy: an adaptable heap visualisation framework. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343–358, New York, NY, USA, 2002.

[PS00] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. *Concurrency: Practice and Experience*, 12(14):1431–1454, 2000.
URL: <`citeseer.ist.psu.edu/depauw99visualizing.html`>.

[RA05] Easwaran Raman and David I. August. Recursive data structure profiling. In *Proceedings of the Third Annual ACM SIGPLAN Workshop on Memory Systems Performance (MSP)*, June 2005.

[Rei03] Steven P. Reiss. Visualizing java in action. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–ff, New York, NY, USA, 2003. ACM Press.

[RR96] Niklas Röjemo and Colin Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 34–41, New York, NY, USA, 1996.

[RR05] Steven P. Reiss and Manos Renieris. Jove: Java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 115–124, New York, NY, USA, 2005.

[SKS00] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. On the effectiveness of GC in Java. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 12–17, New York, NY, USA, 2000.

[SPE98] SPEC Corporation. The SPEC JVM Client98 benchmark suite. `http://www.spec.org/jvm98/jvm98/`, 1998.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
URL: <`citeseer.ist.psu.edu/sagiv96solving.html`>.

[tom] Tom sawyer software. `http://www.tomsawyer.com/home/index.php`.

[WSR00] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.

[yFi] yfiles. `http://www.yworks.com/en/products_yfiles_about.htm`.

[ZZ01] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.