# Tracking Data Structures for Postmortem Analysis (NIER Track)

Xiao Xiao, Jinguo Zhou, Charles Zhang Department of Computer Science and Engineering Hong Kong University of Science and Technology {richardxx, andyzhou, charlesz}@cse.ust.hk

# ABSTRACT

Analyzing the runtime behaviors of the data structures is important because they usually relate to the obscured program performance and understanding issues. The runtime evolution history of data structures creates the possibility of building a lightweight and non-checkpointing based solution for the backward analysis for validating and mining both the temporal and stationary properties of the data structure. We design and implement TAEDS, a framework that focuses on gathering the data evolution history of a program at the runtime and provides a virtual machine for programmers to examine the behavior of data structures back in time. We show that our approach facilitates many programming tasks such as diagnosing memory problems and improving the design of the data structures themselves.

#### **Categories and Subject Descriptors**

D.2.5 [Software Engineering]: Tracing, Debugging aids

#### **General Terms**

Algorithms, Reliability

### Keywords

Tracing, Data Structure, Program Analysis, Debugging

# 1. INTRODUCTION

A large fraction of bugs and program understanding issues are related to data structures. Fred Brooks considered the data structure as the key to understanding a program, and wrote in his famous book, **The Mythical Man-Month**, "show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I will not usually need your flowchart; it will be obvious". Thereby, helping the programmers learn the behaviors of the data structure significantly boosts their productivity.

However, analyzing the data structure statically is difficult and insufficient, especially if we desire the knowledge of

*ICSE* '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

Copyright 2011 ACM 978-1-4503-0445-0/11/05 ...\$10.00.

how the data structures are constructed, evolved and manipulated (*e.g.* data copying), during the program execution. This kind of knowledge, which we call the *data structure evolution history*, can be used to verify the structural properties, to mine anomaly behaviors, to animate the algorithms of the host program, and finally to help the programmers debug and optimize their code.

#### **1.1 Why Data Structure Evolution History?**

The data structure evolution history can be applied and not limited to the following scenarios:

Monitoring. The invariant monitoring is commonly used to guarantee the program correctness at runtime. The conventional approach uses assert to validate the simple value patterns, and checks the structural properties with the programmer supplied code. The periodical execution of the verifying code may decelerate the program dramatically. And perhaps, the slowdown is not tolerable if other higher priority debugging tasks are affected. Moreover, changing the focused invariants requires the re-execution of the program, which is time consuming in some cases. A better approach is dividing the debugging tasks into the online and the offline ones, and performing the offline examination using the data structure updating history. This way naturally separates the different concerns of the debugging tasks.

**Ownership Detection**. An object B is *owned* or *internal* to another object A, if B is dominated by A on the reference graph [8]. The reference graph, from the evolution point of view, is a snapshot of the data structure that captures the connection relationship between objects at a particular time. However, with only a few reference graph snapshots, it is hard to judge whether B is owned by A or not. On the contrary, we have the high confidence to decide if this property holds by inspecting the whole life span of a program execution. The ownership knowledge can be used to help the developers learn the interactions among the high level data structures. For example, in the memory leak detection, knowing two arrays exchanging data is less useful than knowing two hash tables exchanging data, if the arrays are internal to different hash tables.

Memory Leak Detection. In Java, an object is considered *leaked* if it is no longer needed but there are still references to it. Since we only consider the objects with long lifetime as candidates, they are unlikely to be referenced only by local variables. Therefore, there must be some data structures or static fields holding the references to the leaked objects. With the data structure evolution history, we can easily identify the candidate objects by recording the last visiting time of every object. If the dormant period for an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

object is longer than a threshold, we report it as a candidate. Shallow Copy Detection. A common interaction between data structures is data copying. In Java, shallow copy is defined as making a duplicate reference to an object. The knowledge of when the copy occurs and which data structures are involved can facilitate the following program understanding tasks:

- Data Structure Migration. We can add, delete and modify a data structure, we can also migrate a data structure to a new shape. A classical example is the hashmap in JDK. When the container array is nearly full, the resize allocates a new array and copies all the elements to the new container. We call this action migration. A natural usage of the migration knowledge is that, if we know a hashmap always migrates, we can set a larger initial capacity to reduce the overhead incurred by the data copying.
- Copying Bloat Analysis. The copying bloat is a phrase meaning that we copy a large quantities of useless data from one container to another, which is a harmful but widespread phenomenon [16]. Similar to the memory leak detection, if an object is referenced by a new data structure but it is never accessed before that data structure destroyed, it is a suspicious bloat copying. We can also rephrase the methodology in [16] for the evolving data. Suppose we have a pattern language to specify the objects generated by a conceptual *producer* and the ones consumed by the conceptual *consumer*. Now, we scan the whole history and count how many objects generated by the producer are never accessed by the consumer. If the quantity is large, there may be a bloat copying.

Algorithm Animation. A way to learn an algorithm is to study how it manipulates data. For example, the insert routine of the red black tree involves a sophisticated balancing scheme. With the complete history of the insert operation, we can visualize all the intermediate states of the tree and help the programmer understanding.

#### **1.2 Our Contribution Outline**

Our goal is to provide a systematic way to precisely study the data structure evolving history, and to build a platform for a wide range of the data structure related research. To our aim, we present TAEDS (Trace Analysis Engine for Data Structure), a framework for investigating the data structure evolution. TAEDS first parses and instruments the target program, then runs that program to collect the profiling data, *i.e.* the execution trace of interested data structures. After the dynamic phase, it analyzes and reprocesses the obtained trace to facilitate the subsequent analysis tasks. By running the program, we know accurately the shapes and the contents of all the data structures, as well as the side effect of each store statement (*e.q.*  $\mathbf{p}.\mathbf{f} = \mathbf{q}$ ). From the collected trace, we can swiftly reconstruct the snapshot of the data structure at any moment in its lifetime, without running the program again. Moreover, we can navigate forth and back on the trace to change the data structure gradually, which in turn supports any data mining tasks relying on the temporal information.

We have two design guidelines for our TAEDS framework. The first is lightweight in terms of the low runtime overhead and the small trace size, which can be achieved by minimizing the recorded information through static pruning and dynamic compressing techniques. In contrast to the whole program record and replay tools, such as iDNA [1] and the omniscient debugger TOD [13], we only need to record the information that correctly reflects the change to the data structures, which in theory has the dramatically smaller performance penalty. More concretely, some CPU intensive applications prefer to use the local variables in most of their computations and tend to update the data structures infrequently. This case, for iDNA, incurs 13x-15x overhead. For TOD, it comes with 115x slowdown and produces a large trace log (33GB), which is impractical for real use. Therefore, from the pragmatic point of view, tracing only the data structure is a good compromise.

Because we target a less restrictive problem (replaying the data structure, not the whole program), we can build a lightweight *inverse execution trace* (Section 2.2) to quickly roll back the data structure to the last state. However, this technique cannot be efficiently implemented for the universal replayers iDNA and TOD, because an instruction may invoke many implicit memory updates, such as discarding a large bulk of stack data when a function exits (C/C++), or causing the garbage collector to recycle large chunks of memory (Java). Therefore, to support the reverse debugging, iDNA builds many checkpointing frames to help with the backward navigation, which is more expensive than our solution.

The second design guideline is flexibility. We allow the user to selectively monitor the data structure. For example, the user can configure the system to trace only those data structures with the type T and its subtypes. TAEDS is also general enough to handle the multi-threaded programs, by recording the data structure updates for each thread separately and remembering the access order to each shared object, analogous to the iDNA's mechanism [1]. Furthermore, we provide a simple pattern matching language for programmers to select and mark the objects. For example, in the bloat analysis stated above, we are able to tag the objects that are generated by the same static allocation site.

In summary, our system TAEDS is a lightweight and powerful tool to help programmers learn the data structure behaviors. In the next section, we will sketch the TAEDS approach for collecting and compressing the trace log with respect to the single threaded program.

## 2. TRACKING DATA STRUCTURE

TAEDS is a hybrid static and dynamic framework for Java. TAEDS has three components, which are shown in Figure 1. The *indexer* instruments the input program, generates the source code metadata, and monitors the program execution for collecting the raw trace data. The *optimizer* tailors the trace log and builds the inverse execution trace for the subsequent analysis, if needed. The last component, the *simulator*, is both a virtual machine and programming library that contains the primitive functions to assist the programmers in writing the trace analysis code. Next, we describe each of the components with special concern to the indexer.

## 2.1 Trace Indexing

Our principle for the trace collection is that, we do as little as possible while the program is running and try to recover the missed information statically. Following this guideline,

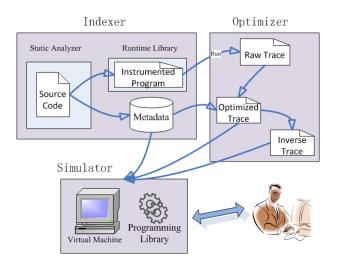


Figure 1: System Architecture of TAEDS

the symbolic names (method name, variable name, types, line numbers, *etc.*) are collected and re-linked to the trace at static time. Only the essential information for guiding the program execution is recorded and processed online.

Our trace log consists of the control flow and the memory access information. The control flow trace indicates the next instruction that would be executed. We choose to implement the Larus's Sequitur algorithm [5] for recording the control flow trace, which has high compression rate and low compress/decompress overhead. The memory access trace is a set of commands to express how the data structures are updated. It records the values written to the instance fields, the array elements and, optionally, the static fields. Next, we introduce our approach to efficiently record and compress the memory access trace.

We first instrument the allocation sites, by putting the instrumentation NewObject(p) immediately after the statement p = new T() of interested types T. The instrumentation writes the *hashCode* of the newly created object to the log file. Next, we run the points-to analysis and gather the pointers  $\mathbb{P}$ , which point to the instrumented objects.

The only statement that changes the traced data structure is the store instruction, e.g. p.f = q where  $p \in \mathbb{P}$ . In the naive approach, we should record both the object pointed to by p and the value of q (may be a primitive type value), which is too redundant because:

1. If p is the pointer this, we only need to record once since it is a constant before a function terminates;

2. We may access the instance fields with the pointer p in many different places, and p is unchanged;

3. The values of p and q may have been recorded before and we do not need to save them twice.

The first issue can be addressed by recording the value of this pointer at the beginning of a non-static function. The second and third issues are handled together through the *dual value predictor*, which depends on the local and global caches. The local cache records the latest value of p (or q), and the global cache stores the last seen K (K > 2) values for each type (*e.g.* int, Object). When we process the store p.f = q, we first compare the current value of p to its local cache. If they are unequal, we update the local cache of p to the current value, and then we look up or insert the global

cache with the current value of  $p^{-1}$  to obtain the position i of the p's value. The variable q is processed in the same way. Depending on the results of the local/global cache hits or misses of p and q, we issue different types of commands to the trace log. For example, if both of the local caches of p and q are hits, we only output a sentinel to indicate the cache hits. If p hits the global cache (so it misses the local cache), we output the position i to the log, otherwise the real value of p is recorded.

The write to the array elements is processed in the same way as the store statements, if distinguishing the positions of the elements in the array is not required. Otherwise, the element offsets are calculated and stored and we disable the local cache to prevent the cache memory blowup. The load statements q = p.f do not need to be processed because they change no values of heap memory. However, in some applications, *e.g.* the memory leak detection, we need to update the last visit time and count the visiting frequency for each object, in which case every load statement is followed by an instrumentation.

Value prediction has shown to be powerful for compressing variable values [2], also demonstrated by Zhang *et.al* [18]. However, more sophisticated value predictors proposed in [2] may not be functional in our problem setting. This is because, the updating history for the heap variables has the weaker data locality compared to the whole program trace. Therefore, the value predictors should be carefully experimented before being adopted in our problem.

#### 2.2 Trace Optimization

The most important task for the *optimizer* is to build the inverse execution trace to enable the backward analysis. The core idea is that, since every command in the trace only slightly updates the data structure, we can roll back to any previous states decrementally. For this purpose, given every command describing a store p.f = q in the trace, we record the value of p.f and generate an inverse command accordingly. For example, the store statement is p.next = null, which aims to nullify the pointer p.next. Suppose p.next points to the object o, we generate an inverse command: p.next = o.

Note that, this strategy works efficiently because we do not need the local pointer information. The same treatment applying to the whole program trace requires the additional effort, because the local pointers should be recovered if we go back to the return statement of a function. The inverse trace is orthogonal to the checkpointing technology as it can be used to help the programmers jump arbitrarily to any moment on the timeline. However, according to our preliminary study, the most common behavior of the programmers is, as for most of the data mining tasks, navigating sequentially on the trace. Therefore, our inverse trace technique is helpful in this common setting.

## 2.3 Trace Simulation

The simulator is a fully featured virtual machine that can reproduce the heap memory states for any moment in the program execution, without presenting the user input again. This is important for the remote debugging setting [12] or when the input is non-reproducible, *e.g.* for a GUI program. To facilitate the interaction with programmers, the simula-

<sup>&</sup>lt;sup>1</sup>It may remove the *least recent visit* value if the cache is full, and the new value inserted is marked *most recent visit*.

tor is designed as a programming library. For example, we can invoke the API to identify the high level conceptual containers through ownership detection. Also, we can search for the objects which are inactive for a long time.

We also implement a garbage collector to help calculate the lifetime of an object. Since we do not care about the values of local pointers, the garbage collector can be invoked after a function terminates. At this point, all the local pointers are set to be null, and an object is not garbage iff it is reachable through some static fields. Therefore, traversing our object reference graph at that moment reclaims the real garbage, and the only side effect is that the computed lifetime of the queried object may be a bit longer.

## 3. RELATED WORK

**Debugging via Memory Snapshots**. Analyzing heap memory snapshots has a long tradition in the area of debugging. Recent work on the memory leak detection [7, 17, 10, 4], the bloat detection [16, 11, 3], the ownership detection and summarization [8], and the source of excessive memory footprints detection [9], all leverages the memory graph to warn the potential performance degeneration. Our work is orthogonal to these approaches. Our data structure evolution trace is aimed at providing the backbone support for these techniques, so that they can be re-implemented in our TAEDS framework with small additional efforts.

Efficient Trace Indexing. The program trace indexing research aims to produce compact trace logs for long running programs, while maintaining the low runtime overhead. The program traces research typically have three directions with respect to the control flow trace [5], the value trace [2], and the dependence trace [14], respectively. The traces can also be highly compressed in combination with some sophisticated scheme, such as the work of Zhang et.al [18]. The traces can be efficiently compressed and stored in other ways. For example, the omniscient debugger OTD [13] incorporates database for the efficient trace retrieval. The whole program replayer iDNA [1] compresses the trace log by eliminating unchanged memory stores, similar to our local cache approach. Venkataramani et.al. even use hardware to achieve the amazingly low (2.7% avg.) runtime overhead [15]. However, all these approaches are about capturing the whole program trace. Since we only focus on the data structure evolution, we can achieve a better result.

Memory Graph Visualization. The automatic software visualization is useful in debugging and education. The compilers, *e.g.* LLVM and GCC, all have the ability to generate the call graph and control flow graph. Research on generating the memory graph is also active, and a good summary is given by [19]. Since our technique compresses the whole data structure evolution history, it is an ideal platform for the software visualization research.

## 4. FUTURE RESEARCH

The idea of offering a query language for the programmers to express their intention recently blossoms in the program analysis area. Since many algorithms share similarities on utilizing the trace (*e.g.* by graph traversal), a facility analogous to  $\mathsf{PQL}$  [6] is particularly useful for programmers who want to quickly examine their observations. For this reason, our future research will explore a declarative way for programmers to work with the trace.

# Acknowledgements

We thank the anonymous reviewers for their insightful feedback. This research is supported by RGC GRF grants 622208 and 622909.

## 5. REFERENCES

- S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06*.
- [2] M. Burtscher. Vpc3: a fast and effective trace-compression algorithm. In SIGMETRICS '04/Performance '04.
- [3] J. Clause and A. Orso. Leakpoint: pinpointing the causes of memory leaks. In *ICSE '10*.
- [4] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL* '07.
- [5] J. R. Larus. Whole program paths. In *PLDI '99*.
- [6] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In OOPSLA '05.
- [7] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing memory leaks using graph mining on heap dumps. In *KDD* '10.
- [8] N. Mitchell. The runtime structure of object ownership. In ECOOP '06.
- [9] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In ECOOP '09.
- [10] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In ECOOP '03.
- [11] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI '09.*
- [12] A. Orso. Monitoring, analysis, and testing of deployed software. In *FoSER '10*.
- [13] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In OOPSLA '07.
- [14] S. Tallam, R. Gupta, and X. Zhang. Extended whole program paths. In PACT '05.
- [15] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *HPCA '07*.
- [16] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: profiling copies to find runtime bloat. In *PLDI '09*.
- [17] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *ICSE* '08.
- [18] X. Zhang and R. Gupta. Whole execution traces. In MICRO '04.
- [19] T. Zimmermann and A. Zeller. Visualizing memory graphs. In SoftVis '01.