EFFECTIVE TECHNIQUES FOR UNDERSTANDING AND IMPROVING DATA STRUCTURE USAGE

A Dissertation
Presented to
The Academic Faculty

by

Changhee Jung

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
August 2013

EFFECTIVE TECHNIQUES FOR UNDERSTANDING AND IMPROVING DATA STRUCTURE USAGE

Approved by:

Dr. Santosh Pande, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Nathan Clark
Virtu Financial

Dr. Hyesoon Kim
School of Computer Science
Georgia Institute of Technology

Dr. Silvius Rus
Quantcast

Dr. Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: June 27, 2013

Soli Deo Gloria

ACKNOWLEDGEMENTS

First and foremost, I would like to thank our Lord and Savior, Jesus Christ, for His grace to carry me through this intense period of my life. Without His intervention and help, it would have been impossible to complete this dissertation. As always, God's grace is sufficient and abundant, and apart from it, I can do nothing.

To my wife, Sangmi. You have been a constant source of encouragement, comfort, and love. Thank you for your sacrifice to live together in Atlanta. It has been four years since I asked you to take a leave of absence from the school in Ann Arbor. I appreciate your patience. You never asked when I would graduate.

I am greatly indebted to my adviser, Prof. Santosh Pande. He offered me many technical challenges to improve my algorithms, and helped me out with great ideas when I was stumped. I have learned so many valuable lessons from him, not only how to find good research topics and solve them analytically, but also how to be a good engineer and mentor.

I would like to thank my ex-advisor, Dr. Nathan Clark, for his excellent guidance and great patience during the first two years of struggling in my Ph.D. study. He deserves thanks for teaching me to understand the high level structure of a problem before jumping into the details.

I owe thanks to the remaining members of my dissertation committee, Prof. Hyesoon Kim, Prof. Sudhakar Yalamanchili, and Dr. Silvius Rus. They all donated their time to help shape this research into what it has become today. I am particularly grateful to Prof. Hyesoon Kim for giving me a great advice for the academic job search and the preparation for the interview. I am also grateful to Dr. Silvius Rus and Dr. Easwaran Raman during my internship at Google.

I also would like to thank current and ex-members of our research lab: Sangho Lee, Kaushik Ravichandran, Tushar Kumar, Jaswanth Sreeram, and Romain Cledat for being great colleagues to work with. I am also thankful to Minjang Kim, Sunpyo Hong, Janghaeng Lee, Jaekyu Lee, Jungju Oh, Moonkyung Ryu, Sangmin Park, Joohwan Lee, and Wonhee Cho, for being great people to discuss the details of my and their work. Finally, to my SFC friends, Hyunshik Shin, Soowon Bae, Hakjoong Kim, Seunghyuk Baek, Youngil Kwon, Youngseok Lee, Jiho Lee, Seunghyun Baek, Jinyong Shim, thank you for the wonderful time we spent together in Christ.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Turing Award winner Niklaus Wirth famously noted, 'Algorithms + Data Structures = Programs', and it follows that data structures should be carefully considered for effective application development. In fact, data structures are the main focus of program understanding, performance engineering, bug detection, and security enhancement, etc. However, due to the nature of ever-changing data structures, their unpredictable performance on program input (underlying computer architecture), and their hard-to-track bug symptoms, existing program analysis techniques have achieved little success.

Our research is aimed at providing effective techniques for analyzing and improving data structure usage in fundamentally new approaches: First, detecting data structures; identifying what data structures are used within an application is a critical step toward application understanding and performance engineering. Through dynamic code instrumentation, our tool can automatically detect the organization of data in memory and the interface functions used to access the data. Then, our dynamic invariant detection determines exactly how those functions modify and utilize the data.

Second, selecting efficient data structures; analyzing data structures' behavior can recognize improper use of data structures and suggest alternative data structures better suited for the current situation where the application runs. This is based on automatically generated machine-learning based models that predict what the best data structure implementation is given a program, a set of inputs, and a target architecture.

Third, detecting memory leaks for data structures; tracking data accesses with little overhead and their careful analysis can enable practical and accurate memory leak detection. To keep the overhead low, we leverage a lightweight monitoring technique based on performance monitoring units available in commodity processors. For accurate memory leak detection, we perform anomaly based statistical analysis.

Finally, offloading time-consuming data structure operations; a dedicated helper thread executes the operations on the behalf of the application thread. By overlapping the executions of both the threads and appropriately synchronizing them, we can take the cost of executing the data structure operations away from the application. In particular, our compiler algorithm automatically eliminates redundant synchronization code misplaced by developers.

CHAPTER I

INTRODUCTION

1.1 The Motivation

Data structures are the main focus of program understanding, performance engineering, bug detection, and security enhancement. Indeed, it is not uncommon to find situations where simply changing the data structures can result in orders of magnitude improvement in application performance for many important domains. For example, scientific applications leveraging matrix inversion [30] and matrix multiplication [141], information mining from large databases [7], and analyzing genetic data for patterns [54], are instances of criticality of data structure selection in an application tuning process. According to [30], proper data structure selection can make the 2-D table implementation used in that study 20 times faster.

However, due to the nature of ever-changing data structures, their unpredictable performance on program input (underlying architecture), and their hard-to-track bug symptoms, existing program analysis techniques have achieved little success. Without effective techniques for data structure analysis and optimization, it would be difficult to improve the applications based on in-depth knowledge on the data structure usage.

In reality, due to the lack of such techniques, developers rarely attempts to understand their data structures. Often times they do not even know what data structure is used in the program they wrote because they simply rely on some standard data structure library. However, data structure libraries were designed to be effective in the common case, and often leave considerable room for improvement in application-specific scenarios.

Besides, the lack of understanding of data structure usage leads developer to unexpected bugs such as memory leaks. Considering many real-world applications allocate their data structures predominantly in heap memory, memory leaks adversely affect the robustness of the applications. In fact, memory leaks are common causes of real-world programming bugs, security breaches, and performance bottlenecks.

The last promising research is to accelerate a critical data structure operation to improve the overall performance of the application. When developers have already selected the best data structure in their application, is it possible to further improve the performance? Prior works have realized some critical data structures with a dedicated hardware logic to make it fast to perform their operations. However, it is questionable if accelerating data structure operations is possible on commodity processors without any hardware support.

1.2 The Contribution

With that in mind, this dissertation seeks to address techniques for analyzing and optimizing data structures in fundamentally new approaches: First, we propose a data structure detection tool called DDT. Detecting data structures can help developers optimize their program; identifying what data structures are used within an application is a critical step toward application understanding and performance engineering. Through dynamic code instrumentation of an application binary, our tool can automatically detect the organization of data in memory and the interface functions used to access the data. Then, our dynamic invariant detection determines exactly how those functions modify and utilize the data.

Second, we propose MIDAS that extends DDT in order to detect data structures even for highly optimized application binaries. Unlike DDT, MIDAS does not rely

on the interface functions of which boundary is eliminated with aggressive compiler optimizations such as function inlining. In particular, this work addresses the difficulty of data structure reasoning in the presence of *destructive updates* [117, 113] when the interface detection is impossible. To this end, MIDAS can accurately identify data structures and extract their useful properties based on the invariants irrespective of how they are encapsulated, how different their implementations are, and even how optimized the binary is.

Third, we propose Brainy the data structure selection tool. Selecting efficient data structures can achieve significant performance gain. Analyzing data structures' behavior can recognize improper use of data structures and suggest alternative data structures better suited for the current situation where the application runs. This is based on automatically generated machine-learning based models that predict what the best data structure implementation is given a program, a set of program inputs, and a target architecture the program is running on.

Fourth, we propose Sniper, an effective memory leak detection tool for C/C++ production software. To track the staleness of allocated memory (which is a clue to potential leaks) with little overhead, Sniper leverages instruction sampling using performance monitoring units available in commodity processors. It neither perturbs the application execution nor increases the heap size. Sniper can deal with even multithreaded applications with very low overhead. In particular, it performs a statistical analysis, which views memory leaks as anomalies, for systematic and automated leak determination.

Lastly, we propose a data structure acceleration technique called DSO. The main idea is to leverage a dedicated thread running on an idle core to offload a time-consuming expensive data structure operation of an application. It is inspired by the previous helper threading approach, i.e., the helper thread executes the data structure operation on the behalf of the application. In this way, DSO can take the

cost of performing the expensive operation away from the application.

1.3 Thesis Statement

The proposed techniques for understanding and improving data structure usage can achieve efficient program execution.

CHAPTER II

DATA STRUCTURE DETECTION

2.1 Introduction

Data orchestration is one of the most critical aspects of developing effective many-core applications. Several different trends drive this movement. First, as technology advances, getting data onto the chip will become increasingly challenging. The ITRS road map predicts that the number of pads will remain approximately constant over the next several generations of processor integration [61]; the implication is that while computational capabilities on-chip will increase, the bandwidth will remain relatively stable. This trend puts significant pressure on data delivery mechanisms to prevent the vast computational resources from starving.

Application trends also point toward the importance of data orchestration. A recent IDC report estimates that the amount of data in the world is increasing ten-fold every five years [50]. That is, data growth is outpacing the current growth rate of transistor density. There are many compelling applications that make use of big data, and if systems cannot keep pace with the data growth then they will miss out on significant opportunities in the application space.

Lastly, a critical limitation of future applications will be their ability to effectively leverage massively parallel compute resources. Creating effective parallel applications requires generating many independent tasks with relatively little communication and synchronization. To a large extent, these properties are defined by how data used in the computation is organized. As an example, previous work by Lee et al. found that effectively parallelizing a program analysis tool required changing the critical data structure in the program from a splay-tree to a

simpler binary search tree [83]. While a splay-tree is generally faster on a single core, splay accesses create many serializations when accessed from multicore processors. Proper choice of data structure can significantly impact the parallelism in an application.

All of these trends point to the fact that proper use of data structures is becoming more and more important for effective manycore software development.

Unfortunately, selecting the best data structure when developing applications is a very difficult problem. Often times, programmers are domain specialists, such as biologists, with no knowledge of performance engineering, and they simply do not understand the properties of data structures they are using. One can hardly blame them; when last accessed, the Wikipedia list of data structures contained *74 different types of trees!* How is a developer, even a well trained one, supposed to choose which tree is most appropriate for their current situation?

Even if the programmer has perfect knowledge of data structure properties, it is still extraordinarily difficult to choose the best data structures. Architectural complexity significantly complicates traditional asymptotic analysis, e.g., how does a developer know which data structures will best fit their cache lines or which structures will have the least false-sharing? Beyond architecture, the proper choice of data structure can even depend on program inputs. For example, splay-trees are designed so that recently accessed items are quickly accessed, but elements without temporal locality will take longer. In some applications it is impossible to know a priori input data properties such as temporal locality. Data structure selection is also a problem in legacy code. For example, if a developer created a custom map that fit well into processor cache lines in 2002, that map would likely have suboptimal performance using the caches in modern processors.

Choosing data structures is very difficult, and poor data structure selection can

have a major impact on application performance. For example, Liu and Rus recently reported a 17% performance improvement on one Google internal application just by changing a single data structure [87]. We need better tools that can identify when poor data structures are being used, and can provide suggestions to developers on better alternatives.

In an ideal situation, an automated tool would recognize what data structures are utilized in an application, use sample executions of the program to determine whether alternative data structures would be better suited, and then automatically replace poor data structure choices.

In this work we attempt to solve the first step of this vision: data structure identification. The **D**ata-structure **D**etection **T**ool, or DDT, takes an application binary and a set of representative inputs and produces a listing of the probable data structure types corresponding to program variables. DDT works by instrumenting memory allocations, stores, and function calls in the target program. Data structures are predominantly stored in memory, and so instrumentation tracks how the memory layout of a program evolves. Memory layout is modeled as a graph: allocations create nodes, and stores to memory create edges between graph nodes. DDT makes the assumption that access to memory comprising a data structure is encapsulated by *interface functions*, that is, a small set of functions that can insert or access data stored in the graph, or otherwise modify nodes and edges in the graph.

Once the interface functions are identified, DDT uses an invariant detection tool to determine the properties of the functions with respect to the graph. For example, an insertion into a linked list will always increase the number of nodes in the memory graph by one and the new node will always be connected to other nodes in the list. A data value being inserted into a splay-tree will always be located at the root of the tree. We claim that together, the memory graph, the set of interface functions, and their invariants uniquely define a data structure. Once identified in

the target application, the graph, interface, and invariants are compared against a predefined library of known data structures for a match, and the result is output to the user. This information can help developers quickly understand their code, particularly if the they are working on a large legacy application, or using shared libraries which may unknowingly be designed poorly. DDT also informs developers of dynamic program properties, such as how effective a hash-function is, and how 'bushy' a tree is, which can be used to optimize the application. DDT could also be as input to performance models that can suggest when alternative data structures may be better suited for an application/architecture.

We have implemented DDT as part of the LLVM toolset [77] and tested it on several real-world data structure libraries: the GNOME C Library (GLib) [132], the Apache C++ Standard Library (STDCXX) [129], Borland C++ Builder's Standard Library implementation (STLport) [127], and a set of data structures used in the Trimaran research compiler [134]. We also demonstrate that DDT works for several real-world applications, enabling the compiler/developer to more easily identify powerful optimizations. This work demonstrates that DDT is quite accurate in detecting data structures no matter what the implementation.

2.2 DDT Algorithm Details

The purpose of DDT is to provide a tool that can correctly identify what data structures are used in an application regardless of how the data structures are implemented. The thesis of this work is that data structure identification can be accomplished by the following: (1) Keeping track of how data is stored in and accessed from memory; this is achieved by building the memory graph. (2) Identifying what functions interact with the memory comprising a data structure; this is achieved with the help of an annotated call graph. (3) Understanding what those functions do; invariants on the memory organization and interface functions are the basis for

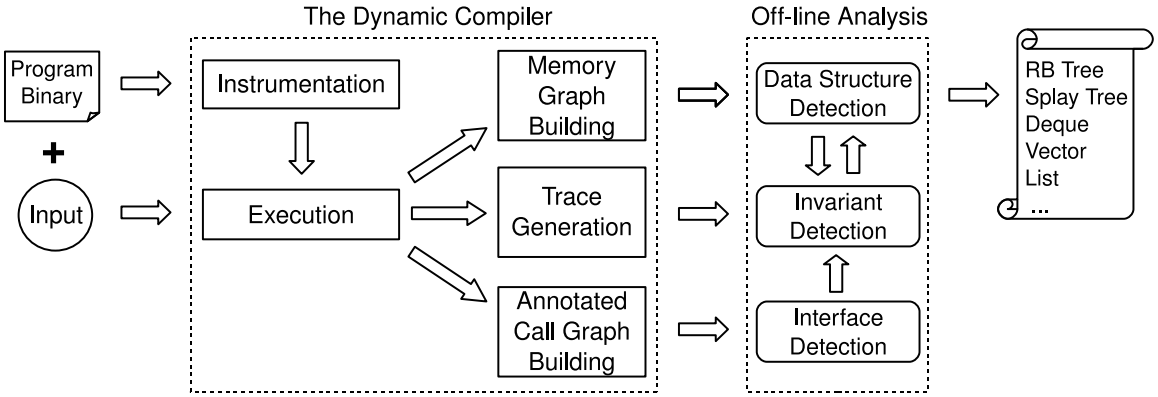


Figure 1: DDT Organization.

characterizing how the data structure operates.

Figure 30 shows a high-level diagram of DDT. An application binary and sample input(s) are fed into a code instrumentation tool, in this case a dynamic compiler. It is important to use sample executions to collect data, instead of static analysis, because static analysis is far too conservative to effectively identify data structures. It is also important for DDT to operate on binaries, because often times data structure implementations are hidden in binary-only format behind library interfaces. It is unrealistic to expect modern developers to have source code access to their entire applications, and if DDT required source code access then it would be considerably less useful.

Once instrumented, the sample executions record both memory allocations and stores to create an evolving memory graph. Loads are also instrumented to determine which functions access various parts of the memory graph, thus helping to delineate interface functions. Finally, function calls are also instrumented to describe the state of the memory graph before and after their calls. This state is used to detect invariants on the function calls. Once all of this information is generated by the instrumented binary, an offline analysis processes it to generate the three traits (memory graph, interface functions, and invariants) needed to uniquely identify a data structure. Identification is handled by a hand-designed

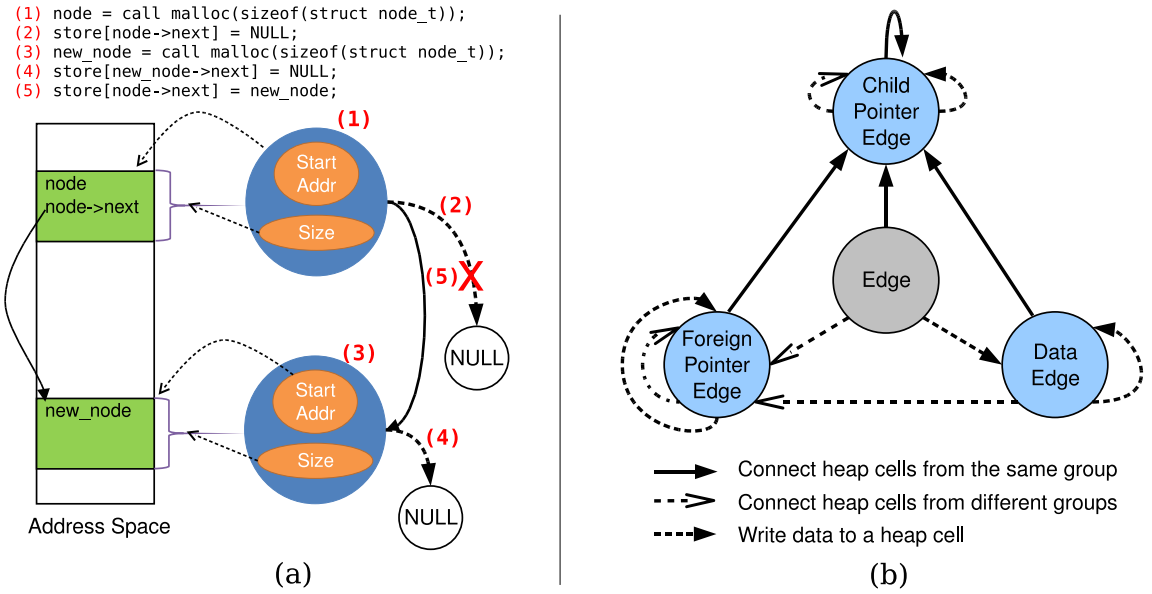


Figure 2: (a) Memory graph construction example. Right side of the figure shows the memory graph for the pseudo-code at top. (b) Transition diagram for classifying edges in the memory graph.

decision tree that tests for the presence of the critical characteristics that distinguish data structures. For example, if nodes in a memory graph always have one edge that points to `NULL` or another node from the same allocation site, and there is an `insert`-like function which accesses that graph, etc., then it is likely that this memory graph represents a singly-linked list. The remainder of this section describes in detail how DDT accomplishes these steps using C++-based examples.

2.2.1 Tracking Data Organization with a Memory Graph

One part of characterizing a data structure involves understanding how data elements are maintained within memory. This relationship can be tracked by monitoring memory regions that exist to accommodate data elements. By observing how the memory is organized and the relationships between allocated regions, it is possible to partially infer what type of data structure is used. This data can be tracked by a graph whose nodes and edges are sections of allocated memory and

the pointers between allocated regions, respectively. We term this a *memory graph*.

The memory graphs for an application are constructed by instrumenting memory allocation functions[1] (e.g., `malloc`) and stores. Allocation functions create a node in the memory graph. DDT keeps track of the size and initial address of each memory allocation in order to determine when memory accesses occur in each region. An edge between memory nodes is created whenever a store is encountered whose target address and data operands both correspond to addresses of nodes that have already been allocated. The target address of the store is maintained so that DDT can detect when the edge is overwritten, thus adjusting that edge during program execution.

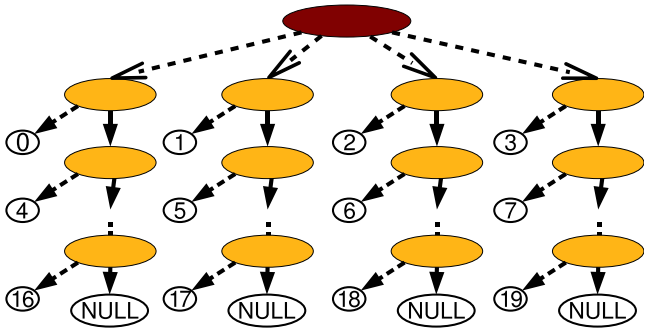
Figure 2 (a) illustrates how a memory graph is built when two memory cells are created and connected to each other. Each of the allocations in the pseudo-code at the top of this figure create a memory node in the memory graph. The first two stores write constant data `NULL` to the offset corresponding to `next`. As a result, two edges from each memory node to the data are created. For the data being stored, two nodes are created. To distinguish data from memory nodes, they have no color in the memory graph. In instruction (5) of the figure, the last store updates the original edge so that it points to the second memory node. Thus, stores can destroy edges between nodes if the portion of the node containing an address is overwritten with a new address. Typically, DDT must simultaneously keep track of several different memory graphs during execution for each independent data structure in the program. While these graphs dynamically evolve throughout program execution, they will also exhibit invariant properties that help identify what data structures they represent, e.g., arrays will only have one memory cell, and

[1]Data structures constructed in the stack, i.e., constructed without explicitly calling a memory allocation routine, are not considered in this work, as it is typically not possible to reconstruct how much memory is reserved for each data structure without access to compiler internals. Custom memory allocators can be handled provided DDT is cognizant of them.

```
vector<list<int> >  v(4);
...

for (int i=0; i < 20; i++)
{
        index = i % v.size();
        v[index].push_back(i);
}
```

(a)



(b)

Figure 3: (a) Code snippet of the program using a `vector` of `list`s and (b) its memory graph.

each node in a binary tree will contain edges to at most two other nodes.

Extending the Memory Graph: The memory graph as presented thus far is very similar to that proposed in previous work [111]. However, we have found that using this representation is not sufficient to identify many important invariants for data structure identification. For example, if the target application contained a singly-linked list of dynamically allocated objects, then it would be impossible to tell what part of the graph corresponded to list and what part corresponded to the data it contains. In order to overcome this hurdle, two extensions to the baseline memory graph are needed: allocation-site-based typing of graph nodes, and typing of edges.

The purpose of allocation-site-based typing of the memory nodes is to solve exactly the problem described above: differentiating memory nodes between unrelated data structures. Many people have previously noted that there is often a many-to-one mapping between memory allocation sites and a data structure type [58]. Thus, if we color nodes in the memory graph based on their allocation site, it is easy to determine what part of the memory graph corresponds to a particular data structure and what part corresponds to dynamically allocated data.

However, in the many-to-one mapping, an allocation site typically belongs to one data structure, but one data structure might have many allocation sites. In

order to correctly identify the data structure in such a situation, it is necessary to merge the memory node types. This merging can be done by leveraging the observation that even if memory nodes of a data structure are created in different allocation sites, they are usually accessed by the same method in another portion of the application. For example, even if a linked-list allocates memory nodes in both `push_front` and `push_back`, the node types can be merged together when a `back` method is encountered that accesses memory nodes from both allocation sites.

While empirical analysis suggests this does help identify data structures in many programs, allocation-site-based coloring does not help differentiate graph nodes in applications with custom memory allocators. That is because multiple data structures can be created in a single allocation site, which is the custom memory allocator. This deficiency could be remedied by describing the custom memory allocators to DDT so that they could be instrumented as standard allocators, such as `malloc`, currently are.

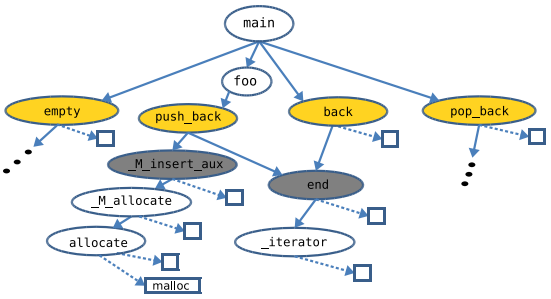
The second extension proposed for the memory graph is typing of edges. As with node coloring, typing the edges enables the detection of several invariants necessary to differentiate data structures. We propose three potential types for an edge in the memory graph: *child*, *foreign*, and *data*. Child edges point to/from nodes with the same color, i.e., nodes from the same data structure. The name "child" edge arose from when we first discovered their necessity while trying to identify various types of trees. Foreign edges point to/from memory graph nodes of different colors. These edges are useful for discovering composite data structures, e.g., `list<set<vector> > >`. Lastly, data edges simply identify when a graph node contains static data. These edges are needed to identify data structures which have important properties stored in the memory graph nodes. E.g., a red-black tree typically has a field which indicates whether each node is red or

black.

A single edge in the memory graph can have several different uses as the dynamic execution evolves, e.g., in Figure 2 (a), the `next` pointer is initially assigned a data edge pointing to `NULL` and later a child edge pointing to `new_node`. The offline invariant detection characterizes the data structure based on a single type for each edge though, thus Figure 2 (b) shows classification system for edges. When a store instruction initially creates an edge, it starts in one of the three states. Upon encountering future stores that adjust the initial edge, the edge type may be updated. For example, if the new store address and data are both pointers from the same allocation site, the edge becomes a child edge, no matter what the previous state was. However, if the edge was already a child edge, then storing a pointer from another allocation site will not change the edge type.

The reason for this can be explained using the example from Figure 2 again. Initially the `next` pointer in a newly initialized node may contain the constant `NULL`, i.e., a data edge, and later on during execution `next` will be overwritten with `new_node` from the same allocation site, i.e., a child edge. Once `next` is overwritten again, DDT can produce more meaningful results if it remembers that the primary purpose of `next` is to point to other internal portions of the data structure, not to hold special constants, such as `NULL`. The prioritization of child edges above foreign edges serves a similar purpose, remembering that a particular edge is primarily used to link internal data structure nodes rather than external data.

Figure 3 gives an example demonstrating why typing nodes and edges in the memory graph is critical in recognizing data structures. The code snippet in this figure creates a `vector` with four `list`s and inserts integer numbers between 0 and 19 into each `list` in a round robin manner. Nodes are colored differently based on their allocation site, and edges types are represented by different arrow structures. To identify the entire data structure, DDT first recognizes the shape



(a)

```
void foo (int num)
{
    // num is mutable
    num = ...

    ...

    v.push_back(num);

}
```

```
template<class T>
void vector<T>::
    push_back(const T& x)
{
    // x is immutable

    ...

    _m_insert_aux(end(), x);

}
```

(b)

Figure 4: (a) Interface identification from a call graph for STL's `vector` and (b) code snippet showing argument immutability.

of a basic data structure for each allocation site by investigating how the "child" edges are connected. Based on the resulting graph invariants, DDT infers there are two types of basic data structures, `vector` and `list`. Then, DDT checks each "foreign" edge to identify the relationship between the detected data structures. In this example, all the elements of `vector` point to a memory node of each `list`, which is a graph invariant. Without the node or edge typing, it would be impossible to infer that this is a composite vector-of-lists instead of some type of tree, for example.

One potential drawback of this approach is that typing of edges and nodes is input dependent, and therefore some important edges may not be appropriately classified. For example, even though an application uses a binary tree, DDT may report it is a linked-list if all the left child pointers of the tree have NULL values due to a particular data insertion pattern. However, our experimental analysis demonstrated no false identifications for this reason, and if a binary tree were behaving as a linked-list, this pathological behavior would be very useful for a developer to know about.

2.2.2 Identifying Interface Functions for the Memory Graph

Understanding how data is organized through the memory graph is the first step toward identifying data structures, but DDT must also understand how that data

is retrieved and manipulated. To accomplish this DDT must recognize what portions of code access and modify the memory graph. DDT makes the assumption that this code can be encapsulated by a small set of *interface functions* and that these interface functions will be similar for all implementations of a particular data structure. E.g., every linked-list will have an insertion function, a remove function, etc. The intuition is that DDT is trying to identify the set of functions an application developer would use to interface with the data structure.

Identifying the set of interface functions is a difficult task. One cannot simply identify functions which access and modify the memory graph, because often one function will call several helper functions to accomplish a particular task. For example, insertions into a `set` implemented as a red-black tree may call an additional function to rebalance the tree. However, DDT is trying to identify `set` functionality, thus rebalancing the tree is merely an implementation detail. If the interface function is identified too low in the program call graph (e.g., the tree rebalancing), the "interface" will be implementation specific. However, if the interface function is identified too high in the call graph, then the functionality may include operations outside standard definitions of the data structure, and thus be unmatchable against DDT's library of standard data structure interfaces.

Figure 4 (a) shows an example program call graph for a simple application using the `vector` class from the C++ Standard Template Library, or STL [125]. In the figure each oval represents a function call. Functions that call other functions have a directed edge to the callee. Boxes in this figure represent memory graph accesses and modifications that were observed during program executions. This figure illustrates the importance of identifying the appropriate interface functions, as most STL data structures' interface methods call several internal methods with call depth of 3 to 9 functions. The lower level functions calls are very much implementation specific.

To detect correct interface functions, DDT leverages two characteristics of interface functions. First, functions above the interfaces in the call graph never directly access data structures; thus if a function does access the memory graph, it must be an interface function, or a successor of an interface function in the call graph. Figure 4 demonstrates this property on the call graph for STL's `vector`. Boxes in this figure represent memory graph accesses. The highest nodes in the call graph that modify the memory graph are colored, representing the detected interface functions.

It should be noted that when detecting interface functions, it is important to consider the memory node type that is being modified in the call graph. That is, if an interface function modifies a memory graph node from a particular allocation site, that function must not be an interface for a different call site. This intuitively makes sense, since the memory node types represent a particular data structure, and each unique data structure should have a unique interface.

You can see that finding the highest point in the call graph that accesses the memory graph is fairly accurate. There is still room for improvement, though, as this method sometimes identifies interface functions too low in the call graph, e.g., `_m_insert_aux` is identified in this example.

The second characteristic used to detect interface functions is that generally speaking, data structures do not modify the data. Data is inserted into and retrieved from the data structure, but that data is rarely modified by the structure itself. That is, the data is, *immutable*. Empirically speaking, most interface functions enforce data immutability at the language-level by declaring some arguments `const`. DDT leverages this observation to refine the interface detection.

For each detected interface function, DDT examines the arguments of those functions that call it and determines if they are modified during the function using

either dataflow analysis or invariant detection. If there are immutable data arguments, then the interface is pushed up one level in the call graph, and the check is repeated recursively. The goal is to find the portion of the call graph where data is mutable, i.e., the user portion of the code, thus delineating the data structure interface.

Using the example from Figure 4, `_m_insert_aux` is initially detected as an interface function. However, its parent in the call graph, `push_back`, has the data being stored as an immutable argument as described in Figure 4 (b). In turn, DDT investigates, its parent, `foo` to check whether or not it is real interface function. Even if `foo` has the same argument, it is not immutable. Thus DDT finally selects `push_back` as an interface function. Detecting immutability of operands at the binary level typically requires only liveness analysis, which is a well understood compiler technique. When liveness is not enough, invariant detection on the function arguments can provide a probabilistic guarantee of immutability. By detecting memory graph modifications, and immutable operands DDT was able correctly to detect that the yellow-colored ovals in Figure 4 (a) are interface functions for STL's `vector`.

One limitation of the proposed interface detection technique is that it can be hampered by compiler optimizations such as function inlining or procedure boundary elimination [133]. These optimizations destroy the calling context information used to detect the interface. Future work could potentially address this by detecting interfaces from arbitrary sections of code, instead of just function boundaries. Source code access would help in this process. A second limitation is that this technique will not accurately detect the interface of data structures that are not well encapsulated, e.g., a class with entirely public member variables accessed by arbitrary pieces of code. However, this situation does not commonly occur in modern applications.

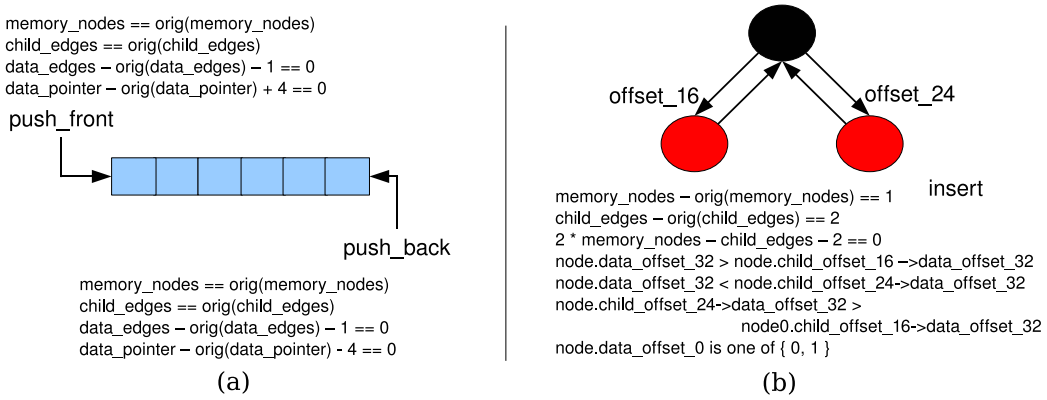


Figure 5: Invariant detection examples of interface functions; (a) STL *deque*, (b) STL *set*.

2.2.3 Understanding Interface Functions through Invariant Detection

Now that the shape of the data structure and the functions used to interface with the data are identified, DDT needs to understand exactly what the functions do, i.e., how the functions interact with the data structure and the rest of the program. Our proposed solution for determining what an interface function does is to leverage dynamic invariant detection. Invariants are program properties that are maintained throughout execution of an application. For example, a min-heap will always have the smallest element at its root node or a data value being inserted into a splay-tree will always become a new root in the tree. Invariants such as these are very useful in many aspects of software engineering, such as identifying bugs, and thus there is a wealth of related work on how to automatically detect probable invariants [49].

Invariant properties can apply before and after function calls, e.g., `insert` always adds an additional node to the memory graph, or they can apply throughout program execution, e.g., nodes always have exactly one child edge. We term these *function invariants* and *graph invariants*, respectively. As described in Section 2.2.1,

graph invariants tell DDT the basic shape of the data structure. Function invariants allow DDT to infer what property holds whenever functions access the data structure as the example.

In using invariants to detect what data structures are doing, DDT is not concerned so much with invariants between program variables as much as it is concerned with invariants over the memory graph. For example, again, insertion to a linked list will always create a new node in the memory graph. That node will also have at least two additional edges: one pointing to the data inserted, and a next pointer. By identifying these key properties DDT is able to successfully differentiate data structures in program binaries.

Target Variables of Invariant Detection: The first step of invariant detection for interface functions is defining what variables DDT should detect invariants across. Again, we are primarily concerned with how functions augment the memory graph, thus we would like to identify relationships of the following variables before and after the functions: *number of memory nodes*, *number of child edges*, *number of data edges*, *value pointed by a data edge*, and *data pointer*. The first three variables are used to check if an interface is a form of insertion. The last two variables are used to recognize the relationship between the data value and the location it resides in, which determines how the value affects deciding the location that accommodates it.

As an example, consider the STL `deque`'s[2] interface functions, `push_front` and `push_back`. DDT detects interesting invariant results from the target variables mentioned above, as shown on the left side of Figure 5. Since the STL `deque` is implemented using dynamic array, *number of memory nodes* and *number of child*

[2] `deque` is similar to a vector, except that it supports constant time insertion at the front or back, where vector only supports constant time insertion at the back.

edges remain unchanged when these interface functions are called. DDT recognizes that these interface functions insert elements; however, because *number of data edges*, represented as 'data_edges' in the figure, increase whenever these functions are called. In the `push_front`, *data pointer* decreases while it increases in the `push_back`, meaning that data insertion occurs in head and tail of the `deque`, respectively. That lets us know this is not an STL `vector` because `vector` does not have the `push_front` interface function.

The right side of Figure 5 shows another example of the seven invariants DDT detects in STL `set`'s interface function `insert`. The first two invariants imply that the `insert` increases *number of memory nodes* and *number of child edges*. That results from the fact the `insert` creates a new memory node and connects it to the other nodes. In particular, the third invariant, "*2 * number of memory nodes - number of child edges - 2 == 0*," tells us that every two nodes are doubly linked to each other by executing the `insert` function. The next three invariants represent that the value in a memory node is always larger than the first child and smaller than the other child. This means the resulting data structure is a similar to a binary tree. The last invariant represents that there is a data value that always holds one or zero. STL `set` is implemented by using red-black tree in which every node has a color value (red or black), usually represented by using a boolean type.

Similar invariants can be identified for all interface functions, and a collection of interface functions and its memory graph uniquely define a data structure. In order to detect invariants, the instrumented application prints out the values of all relevant variables to a trace file before and after interface calls. This trace is post-processed by the Daikon invariant detector [49] yielding a print out very similar to that in Figure 5. While we have found invariants listed on the graph variables

defined here to be sufficient for identifying many data structures, additional variables and invariants could easily be added to the DDT framework should they prove useful in the future.

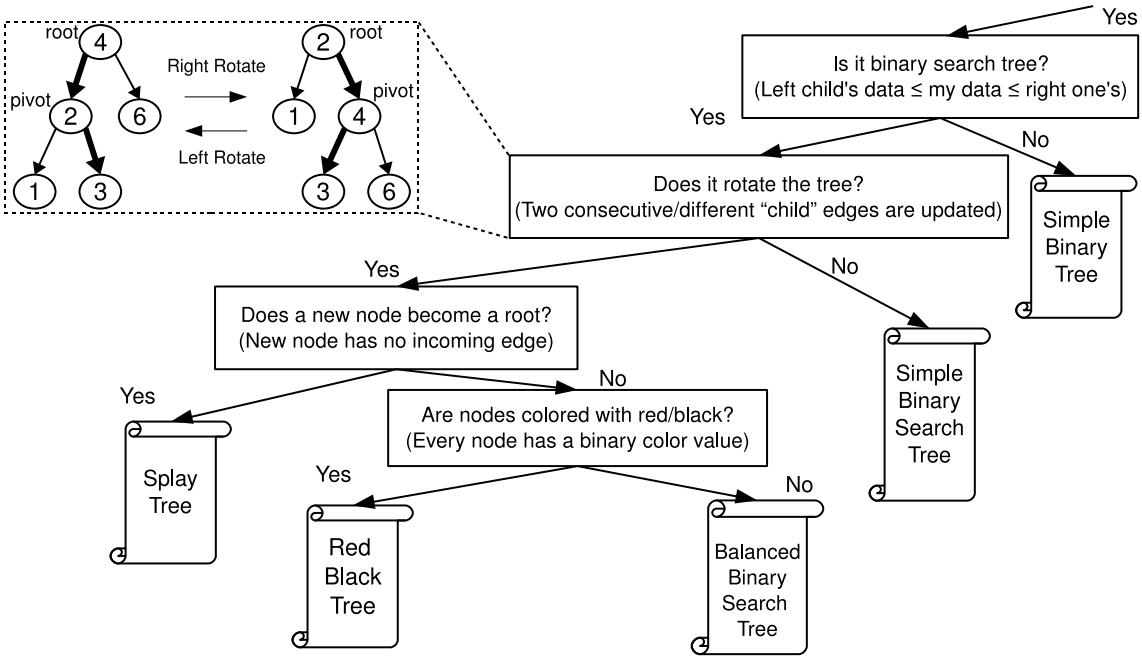


Figure 6: Portion of the decision tree for recognizing binary search trees in DDT.

2.2.4 Matching Data Structures in the Library

DDT relies on a library of pre-characterized data structures to compare against. This library contains memory graph invariants, a set of interface functions, and invariants on those interface functions for each candidate data structure. The library is comprised of a hand-constructed decision tree that checks for the presence of critical invariants and interface functions in order to declare a data structure match. That is, the presence of critical invariants and interface functions is tested, and any additional invariants/interfaces to not override this result.

The invariants are picked that distinguish essential characteristics of each data structure, based on its definition rather than on implementation. That is, for a

linked list, the decision tree attempts to look for an invariant, "*an old node is connected to a new node*" instead of "*a new node points to* `NULL` ". The latter is likely to be implementation specific. Intuitively, the memory graph invariants determine a basic shape of data structures, e.g., each node has two child edges. Meanwhile, the invariants of interface functions distinguish between those data structures which have similar shapes. Extending this library is an easy process: simply run a sample execution of an application with the target data structure, look through the list of identified invariants, and add the critical invariants into the decision tree. In practice, a new data structure can be added to the library in a few minutes.

At the top of the decision tree, DDT first investigates the basic shape of data structures. After the target program is executed, each memory graph that was identified will have its invariants computed. For example, an STL `vector` will have the invariant of only having a single memory node. With that in mind, DDT guesses the data structure is `array`-like one. This shape information guides DDT into the right branch of the decision tree in the next to check desired function invariants.

Among the detected interface functions, DDT initially focuses on `insert`-like functions. That is because most data structures have at minimum an insertion interface function, and they are very likely to be detected regardless of program input. If the required interface are not discovered, DDT reports that the data structure does not match. After characterizing the insertion function, DDT further investigates other function invariants traversing down the decision tree to refine the current decision. As an example, in order to determine between `deque` and `vector`, the next node of the decision tree investigates if there is the invariant corresponding to `push_front` as shown in Section 2.2.3. It is important to note that the interface functions in the library contain only *necessary* invariants. Thus if the dynamic invariant detection discovers invariants that resulted only because of

unusual test inputs, DDT does not require those conservative invariants to match what is in the library.

Figure 6 shows a portion of DDT's decision tree used to classify binary trees. At the top of the tree, DDT knows that the target data structure is a binary tree, but it does not know what type of binary tree it is. First, the decision tree checks if there is the invariant corresponding to a "binary search tree". If not, DDT reports that the target data structure is a simple binary tree. Otherwise, it checks if the binary tree is self-balancing. Balancing is implemented by tree rotations and they are achieved by updating child edges of *pivot* and *root*, shown in the top-left of Figure 6. The rotation function is detected by the invariant that two consecutive and different "child" edges are overwritten (shown in bold in Figure 6). If tree-rotation is not detected in the `insert`, DDT reports that the data structure is a "simple binary search tree." More decisions using the presence of critical functions and invariants further refine the decision until arriving at the leaf of the decision tree, or a critical property is not met, when DDT will report an unknown data structure. After data structures are identified, the decision tree is repeated using any "foreign" edges in the graph in order to detect composite data structures, such as `vector<list<int> >`.

Using invariant detection to categorize data structures is probabilistic in nature, and it is certainly possible to produce incorrect results. However, this approach has been able to identify the behavior of interface functions for several different data structure implementations from a variety of standard libraries, and thus DDT can be very useful for application engineering. Section 4.6 empirically demonstrates DDT can effectively detect different implementations from several real-world data structure libraries.

2.3 Evaluation

In order to demonstrate the utility of DDT, we implemented it as part of the LLVM toolset. DDT dynamically instruments the LLVM intermediate representation (IR), and the LLVM JIT converts the IR to x86 assembly code for execution. Output from the instrumented code is then fed to Daikon [49] to detect invariants needed to identify data structures. These invariants are then compared with a library of data structures that was seeded with simple programs we wrote using the C++ Standard Template Library (STL) [125]. The entire system was verified by recognizing data structures in toy applications that we wrote by hand without consulting the STL implementation. That is, we developed the classes `MyList`, `MySet`, etc. and verified that DDT recognized them as being equivalent to the STL implementations of `list`, `set`, etc. Additionally, we verified DDT's accuracy using four externally developed data structure libraries: the GNOME project's C-based GLib [132], the Apache C++ Standard Library STDCXX [129], Borland C++ Builder's Standard Library STLport [127], and a set of data structures used in the Trimaran research compiler [134].

Even though the current implementation of DDT operates on compiler IR, there is no technical issue preventing DDT's implementation on legacy program binaries. The LLVM IR is already very close to assembly code, with only two differences worth addressing. First, LLVM IR contains type information. The DDT tool does not leverage this type information in any way. Second, LLVM IR is not register allocated. The implication is that when DDT instruments store instructions it will avoid needlessly instrumenting spill code that may exist in a program binary. This may mean that the overhead experienced for instrumentation is probably underestimated by a small factor. It is likely to be a small factor, though, because the amount of spill code is generally small for most applications.

2.3.1 Demonstrating the Correctness of DDT

Table 1: Data structure detection results of representative C/C++ data structure libraries.

Library	Data structure type	Main data structure	Reported data structure	Identified?
STL	vector	dynamic array	vector	yes
	deque	double-ended dynamic array	deque	yes
	list	doubly-linked list	doubly-linked list	yes
	set	red-black tree	red-black tree	yes
Apache (STDCXX)	vector	dynamic array	vector	yes
	deque	double-ended dynamic array	deque	yes
	list	doubly-linked list	doubly-linked list	yes
	set	red-black tree	red-black tree	yes
Borland (STLport)	vector	dynamic array	vector	yes
	deque	double-ended dynamic array	deque	yes
	list	doubly-linked list	doubly-linked list	yes
	set	red-black tree	red-black tree	yes
GLib	GArray	double-ended dynamic array	deque	yes
	GQueue	doubly-linked list	doubly-linked list	yes
	GSList	singly-linked list	singly-linked list	yes
	GTree	AVL tree	balanced binary search tree	no
Trimaran	Vector	dynamic array	vector	yes
	List	singly-linked list	singly-linked list	yes
	Set	singly-linked list	singly-linked list	yes

Table 1 shows how DDT correctly detects a set of data structures from STL, STDCXX, STLport, GLib, and Trimaran. The data structures in this table were chosen because they represent some of the most commonly used, and they exist in most or all of the libraries examined (there is no tree-like data structure in Trimaran). Several synthetic benchmarks were used to evaluate DDT's effectiveness across data structure implementations. These benchmarks were based on the standard container benchmark [12], a set of programs originally designed to test the relative speed of STL containers. These were ported to the various data structure libraries and run through DDT.

Overall, DDT was able to accurately identify most of the data structures used in those different library implementations. DDT correctly identified that the `set` from STL, STDCXX, STLport were all implemented using a red-black tree. To accomplish this, DDT successfully recognized the presence of tree-rotation functions, and that each node contained a field which contains only two values: one for "red"

and one for "black". DDT also detected that Trimaran's `Set` exploits list-based implementation and GLib's `GQueue` is implemented using a doubly-linked list.

The sole incorrect identification was for GLib's `GTree`, which is implemented as an AVL tree. DDT reported that it was a balanced binary search tree because DDT only identified that there are invariants of tree-rotations. In order to correctly identify AVL trees, DDT must be extended to detect other types of invariants. This is a fairly simple process, however, we leave this for future work.

On average, the overhead for instrumenting the code to recognize data structures was about 200X. The dynamic instrumentation overhead for memory/call graph generation was about 50X while the off-line analysis time including interface identification and invariants detection occupies the rest of the overhead. In particular, the interface identification time was sufficiently negligible that it occupies less than 3% of the whole overhead. While this analysis does take a significant amount of time, it is perfectly reasonable to perform heavy-weight analysis like this during the software development process.

2.3.2 Demonstrating the Utility of DDT

DDT helps programmers understand and optimize their applications by identifying the critical data structures within applications. The introduction described a motivation of automatically replacing data structures in parallel applications, but many other optimizations are enabled by this analysis, e.g., data structure aware prefetching. Below, we describe an empirical study of using DDT to help optimize six applications. All the experiments were performed on a Linux-based system with a 2.33 GHz Intel Core2 Quad CPU and 4GB of RAM.

2.3.2.1 Em3d

Em3d is a benchmark from the Olden Benchmark Suite [3] that computes electromagnetic field values in a 3D space. It maintains two linked-lists that represent

electric and magnetic fields. Its hot loop traverses each node in one list, computes a result value by performing convolution of the node's scaling vector and stores the value to nodes in the other list. A critical property of this applications is that the resulting value is only written to the current node, which means it does not cause data dependence on the next iteration's computation. The singly linked-list is recognized by DDT. Based on invariants on its interface functions, DDT found out that inserting data is occurred at the end of the linked-list. In other words, DDT detects that the linked lists in this application can be replaced with a vector-like dynamic array, which improves data locality, thereby achieving a speedup of 1.14. Replacing the linked list with a vector also enables well-known automatic parallelization transformations that do not work on linked lists. By manually parallelizing the critical loop, we quickly achieved a speedup of 1.59 for this application.

2.3.2.2 Bh

Bh, also from the Olden Suite, performs a *Barnes and Hut* N-body force algorithm on the gravitational interaction. To access the bodies, the program maintains a linked-list. The main computation is occurred in a loop in *grav* function, which reads each body and traverses a space decomposition tree from its root to compute the body's new acceleration value. Similar to em3d, each resulting value written is never read in the critical loop, thereby causing no data dependence on any other body nodes. DDT again reported its data structure as a singly linked-list that could be replaced by a vector. Replacing the linked-list with a vector yielded a speedup of 1.34, and manually simulating well-understood automatic parallelization of the hot loop in *grav*, we finally obtained a speedup of 4.35 on our 4-core machine.

2.3.2.3 Raytrace

Raytrace draws a 3D image of groups of spheres using ray tracing algorithm implemented in C++. The spheres are divided into groups that use a linked list to

store them. The main computation of the program occurred in a loop in *intersect* of each group object. First, the intersection calculation is performed for each group of spheres. If a ray hits the group, it is subsequently performed for its spheres (scenes). DDT correctly reported its data structure as a doubly linked-list and found that data is inserted at the end of the linked-list. Replacing the linked-list with a vector again yielded a speedup of 1.17. The hot loop does exhibit do-all style parallelism, however, it is not as simple to parallelize as the previous examples. Instead of parallelizing it, we injected software prefetch instructions in the loop body by hand, using knowledge of the data structure. One interesting information DDT reported, the original linked list keeps a pointer to another heap allocated object as a data value. In other words, the replaced data structure should be `vector` of pointers. With that in mind, for the prefetch target address, we used data value itself of the `vector`, not data index location. This is novel, effective prefetching strategy compared to other statistics based prefetching techniques, which mostly do not work for irregular memory access patterns. By applying the data structure conscious prefetching technique, we achieved a final speedup of 1.38.

2.3.2.4 Xalancbmk

Xalancbmk is an XSLT processor that performs an XML to HTML transformations. It takes as inputs an XML document and an XSLT stylesheet file that contains detailed instructions for the transformation. The program maintains a string cache comprised of two levels, *m_busyList* and *m_availableList*, `vector`s. When a string is freed in *XalanDOMStringCache::release*, it moves the string to the *m_availableList* provided it is found in the *m_busyList*. DDT correctly recognized that the both string lists are implemented using `vector`s and reported that the program contains several red-black trees. Interestingly, the invariants of one interface function for the *vector* invoked by *XalanDOMStringCache::release* describe that the interface

function loads a sequence of addresses with the same stride from the start address of the `vector`, which is exactly what *std::find* does for `vector`. This was suspicious enough for us to suspect that it performs a linear search having O(n) time complexity. We replaced the data structure with STL `set` in which the searching operation uses a binary search algorithm, i.e., O(log n). This transformation achieved a speedup of 1.13.

2.3.2.5 Jaula

Jaula is a C++ based JSON parser implemented using STL. It verifies the syntax of a JSON file and writes the reduced expression of each JSON type instance as an output to cut down the file size. During the parsing, the program creates various JSON type instances based on a lexical analysis result. The instances are stored in different data structures and all their elements are iterated to generate the output. DDT correctly recognized that the two main JSON type instances, *object* and *array*, are maintained using a red-black tree and a doubly-linked list, respectively. In particular, DDT reported their insert-like interface functions as *array::addItem* and *object::insertItem* differently. The reason is that DDT performs argument-immutability based interface detection. However, since these are just wrappers of STL interface functions, *list*$< T >$*::push_back* and *map*$< T >$*::insert*, which means their invariant results are identical, DDT could correctly identified such data structures. We replaced the linked list with `vector` as its original name (*array*) implies, however, we did not get a significant speedup. This results from the fact that the syntax of JSON documents is quite simple, and therefore the majority of the execution time is spent on the lexical analysis.

2.3.2.6 DDT Memory Graph

Here DDT itself was used as a test application. At runtime, DDT keeps detailed information about dynamically allocated memory chunks, e.g., memory nodes to

keep track of which two of them are connected to each other. This information is stored using an STL *set* implemented as a red-black tree. DDT correctly recognized this critical data structure and found out that an interface function accessing it, `operator++`, is invoked much more frequently than the insert-like function. This tells us that the benchmark spends the majority of its execution time on iterating the data structure repeatedly. This was the case because on every memory operation, the tree needed to be iterated to determine if the address modified affected memory graph nodes. Thus, DDT tells us that a data structure with more efficient lookup is appropriate for implementing this particular set. We implemented a new version of the `map` that can lookup all memory nodes whose range contains a target address in O(log N) time, instead of the O(N) version that was previously implemented. Replacing this data structure, DDT was able to profile 181.mcf from SPECint 2000 in just 6 seconds, where previously it took over 24 hours to complete.

These examples show that DDT can be used to help developers understand and easily optimize their applications. This can take the form of identifying replacement data structures, or enabling other optimizations such as automatic parallelization or data prefetching.

2.4 Summary

The move toward manycore computing is putting increasing pressure on data orchestration within applications. Identifying what data structures are used within an application is a critical step toward application understanding and performance engineering for the underlying manycore architectures. This work presents a fundamentally new approach to automatically identifying data structures within programs.

Through dynamic code instrumentation, our tool can automatically detect the organization of data in memory and the interface functions used to access the data.

Dynamic invariant detection determines exactly how those functions modify and utilize the data. Together, these properties can be used to identify exactly what data structures are being used in an application, which is the first step in assisting developers to make better choices for their target architecture. This paper demonstrates that this technique is highly accurate across several different implementations of standard data structures. This work can provide a significant aid for assisting programmers in parallelizing their applications. We plan future work to extend DDT by integrating cost models to predict when alternative data structures are better suited for the target application, and providing semi-automatic or speculative techniques to automatically replace poorly chosen data structures.

CHAPTER III

DATA STRUCTURE DETECTION WITHOUT INTERFACE FUNCTIONS

3.1 Introduction

Understanding how data is stored and accessed in programs is a critical aspect of the design and maintenance of software. With a knowledge of what data structures are being utilized, a developer can more easily maintain code [37], reverse engineer an application [110], find bugs [65], and even enforce data-structure-specific consistency properties to make the application more secure and stable [39]. Additionally, recent studies show that identifying data structures is very useful for malware detection [107] as well as memory leak detection [147].

High-level information on data structures also enables new types of optimization strategies. Data structure aware compilers improve the quality of alias analysis, thereby achieving more efficient lock generation for user-specified atomic regions [136]. Using the specificity of underlying data structures, data structure libraries can integrate a prefetching thread to leverage data access patterns [90]. Memory allocators take advantage of data structure knowledge as hints for improving memory reference locality [64]. Similarly, high-level information about data structures significantly improves techniques for data object layout [29] and *pool allocation* [78].

The industry trend toward manycore processors has shifted the burden of improving system performance primarily from hardware vendors to software engineers and a critical component of software performance is an application's ability to effectively leverage parallel compute resources. Parallelization of an algorithmic

step at the level of a function call is very useful for converting sequential semantics into optimistic parallelism. However, optimistic parallelism at function level can high overheads due to rollbacks and thus, high confidence in optimism is essential for effective parallelization at the function level. Knowledge of the underlying data structure (especially its shape in terms of whether it is a tree or a graph with many joins) can effectively serve to boost the confidence for aggressive optimistic parallelization. Programming models such as Galois [74] and languages such as Deterministic Parallel Java [14] allow the programmers to leverage data structure properties for parallelization. It is also shown that the careful selection of data structures can significantly impact the amount of parallelism in an application [9, 86, 22].

For these reasons and many more [30, 141, 7], tools that automatically identify the data structures in a program can contribute significantly to the effective software development and optimization. If we do not know what data structures is used, how can we enable data structure aware optimizations? Unfortunately, it is very difficult to identify what data structures are utilized in programs. *Data structure implementations are often provided exclusively in binary form and hidden behind library interfaces.* Even when complete source files are available, implementation idiosyncrasies can complicate the process of identifying data structures. Due to these complications, it becomes an overwhelming challenge for software developers to recognize that different implementations possess the same fundamental properties.

This paper presents the design and implementation of MIDAS, a framework for data structure identification. To effectively identify data structures, MIDAS leverages dynamic analysis on an application binary. Memory allocation functions

and load/store instructions are instrumented using dynamic binary instrumentation. The instrumentation code monitors how the memory layout of the application evolves during program execution. To keep track of the memory layout, MIDAS dynamically constructs and updates a *memory graph* where heap-allocated objects and their points-to relations are represented by vertices and edges, respectively. Invariant properties of the graph and data values of a data structure are then matched against a library of known data structures, providing a probabilistic identification.

Unfortunately, the invariants may be violated at times, which makes it impossible to identify data structures correctly. The reason for such violations is twofold; (1) data structures suffer from *destructive updates* when they are updated [117, 113]. (2) data structures may take on various forms at different times during execution, e.g., a binary tree can look like a linked list according to particular data insertion/deletion patterns. The execution traces in both cases easily falsify the critical invariants of the data structure necessary for proper identification.

The insight to the problem is that when data structures lose the defined shape, their critical invariants do not hold. In other words, if data structures retains the defined shape, their critical invariants should hold. With that in mind, MIDAS figures out the defined shape of a data structure to filter out those traces generated while it loses the shape. The challenge is how to recognize the defined shape in the presence of constant changes of the data structre shape.

To achieve this, MIDAS relies on the observation that for most of the time, a data structure shows the defined shape. That leads MIDAS to take an inductive manner to catch the shape; MIDAS associates each trace with *memory graph magnitude* which coarsely but effectively summarizes the ever-changing shape of data strucures over time. Then, MIDAS post-processes the collection of execution traces and filters out problematic traces which possess rare *memory graph magnitudes*. This

ensures that only essential traces generated while data structures retain the defined shape are used into the invariant detector for their identification.

In particular, MIDAS detects not only shape (structural) but also data (numerical) invariants of data structures—effectively differentiating those data structures that possess very similar or identical shapes, e.g., discerning a binary search tree from a binary tree. Experimental results show that MIDAS can accurately identify data structures and extract their useful properties based on the invariants irrespective of how they are encapsulated, how different their implementations are, and even how optimized the binary is.

3.2 MIDAS: Mining Data Structures

The purpose of MIDAS is to provide a tool that can correctly identify the data structures that are used in an application regardless of how the data structures are implemented, how they are encapsulated, and even how they are optimized in an application binary. MIDAS is an automated approach requiring no user intervention. There are three primary tasks performed by MIDAS in order to facilitate accurate data structure identification; (1) Keeping track of how data is stored in and accessed from memory; this is achieved by building a *memory graph*. (2) Recording the signature of a data structure; invariants on the shape and data values of a data structure provide a unique identifier. (3) Filtering out execution traces generated while data structures lose the characteristics shape thereby violating their critical invariants.

The high-level workflow of MIDAS is as follows. First, application binary is fed into dynamic binary instrumentation tool. To track the memory layout of individual data structures, memory allocation functions are dynamically instrumented

together with load/store instructions that access heap memory. The instrumentation code interacts with MIDAS runtime. It maintains *memory graphs* of data structures and records their execution traces to a file to facilitate the detection of shape and data invariants during later stages. It is important for MIDAS to operate on binaries, because data structure implementations are often hidden in binary-only formats behind library interfaces. Besides, it is unrealistic to expect developers to have source code access to their entire applications, and—accordingly– if MIDAS required source code access, it would be considerably less useful.

Once program execution finishes, MIDAS analyzes the trace files and selects only the critical execution traces of data structures. This is very important because—at times— data structures may not show their defined shapes due to *destructive updates* and data structure change. For example, insertion into (or removal from) the middle of a doubly-linked list makes it temporarily lose the characteristic appearance of a doubly-linked list. A similar problem arises in the case of tree rotation of a binary search tree. Even worse, a data structure may change its shape from one to another, e.g., a singly-linked list can change to a doubly-linked or even a binary tree. Therefore, those execution traces generated while a data structure has temporarily lost its defined shape easily falsify its critical invariants and—as a result—prohibit proper identification.

To overcome these challenges, MIDAS attempts to capture the defined shape of a data structure in an inductive manner using its *memory graph magnitude* which effectively summarizes the essence of the shape. This approach is based on the observation that data structures will demonstrate their characteristic shape during the majority of the execution time. Thus, MIDAS post-processes the collection of execution traces and filters out problematic traces that possess a rare *memory graph magnitude*, which ensures that only essential traces are fed into the invariant detector. Once the shape and data invariants of a data structure are detected, they are

compared against a predefined library of known data structures. E.g., if the shape invariant says nodes in a memory graph always have one edge that points to `NULL` or another node from the same allocation site, and the data invariant says data values in each node are always less than the values in their successor nodes, then MIDAS reports the data structure as a sorted, singly-linked list. The remainder of this section describes in detail how MIDAS accomplishes these steps.

3.2.1 Tracking Data Organization

The first task of characterizing a data structure involves understanding how data elements are maintained within memory. This relationship can be tracked by monitoring memory regions that exist to accommodate data elements. By observing how the memory is organized and the relationships between allocated regions, it is possible to partially infer what type of data structure is used. This data can be tracked by using a *memory graph*, which is a directed graph with heap-allocated objects as vertices and points-to relations as edges. For example, in the graph, there is an edge from vertex a to vertex b if the object corresponding to a points to the object corresponding to b.

The memory graphs for an application are constructed by instrumenting memory allocation functions (e.g., `malloc`) [1] and stores. Allocation functions create a node in the *memory graph*. MIDAS keeps track of the size and initial address of each memory allocation in order to determine when a memory access occurs in each region. An edge between memory nodes is created whenever a store is encountered whose target address and data operands both correspond to addresses of nodes that have already been allocated. The target address of the store is maintained so that MIDAS can detect when the edge is overwritten, thus adjusting that edge during program execution. While these graphs dynamically evolve throughout

[1] This work assumes that MIDAS is aware of non-standard memory allocators.

program execution, they will also exhibit shape invariants that help identify the data structures they represent, e.g., each node in a binary tree will contain edges to at most two other nodes.

It needs to be noted that multiple data structures can be contained in a memory graph. To differentiate data structure instances in the memory graph, each node is colored based on its allocation site. With the help of this node coloring scheme, MIDAS can identify the recursive backbone of each data structure and detect its shape invariants. Again, the shape invariants identify data structures with different shapes.

However, many data structures have very similar or identical shapes and, consequently, they also possess the same shape invariants. To differentiate such data structures, MIDAS also leverages the *data* invariants of a data structure, which may, for example, discern a binary search tree from a binary tree. Detecting the data invariants of a data structure requires the typing of edges in the memory graph. Each edge in the memory graph is typed as one of three edge classes: *child*, *foreign*, or *data*. Child edges point to/from nodes with the same color, i.e., nodes from the same data structure. The name "child" edge arose from when we first discovered their necessity while trying to identify various types of trees. Foreign edges point to/from memory graph nodes of different colors. These edges are useful for discovering composite data structures, e.g., a tree of linked-lists. Lastly, data edges simply identify when a graph node contains static data. These edges are needed to identify data structures which have important properties stored in the memory graph nodes. E.g., a red-black tree typically has a field which indicates whether each node is red or black.

3.2.2 Recording the Signature of a Data Structure

The critical step needed to accomplish data structure identification is to record the signature of a data structure. Invariant properties on the shape and the data values of a data structure are the basis for characterizing different data structures. To achieve this, MIDAS leverages dynamic invariant detection. MIDAS records the execution trace of a data structure to a file and post-processes the trace to detect the shape and data invariants of the data structure. The first step of invariant detection is defining what variables MIDAS should detect invariants across.

The target variables as follows; (1) *number of memory nodes*, (2) *number of child edges*, (3) *address pointed to by child edges*, (4) *address pointed to by foreign edges*, (5) *value held in a data edge*. The first two variables are necessary to calculate the *memory graph magnitude* metric (see Section 3.2.2.2). Note that they are maintained for each connected component of the memory graph so that the metric is computed per connected component per data structure instance. (2)-(4) detect structural invariants of data structures such as 'prev_ptr(next_ptr(my_node)) == my_node' in a doubliy-linked list.

Again, (4) is necessary for discovering compositie data structures. E.g., to identify a tree of linked-lists, MIDAS first recognizes the shape of a basic data structure for each allocation site by investigating how the child edges are connected. Based on the resulting invariants, MIDAS infers there are two types of basic data structures, tree and list. Then, MIDAS checks each foreign edge to identify the relationship between the detected data structures. Here, all the tree nodes point to a memory node of each list, which is another invariant. Finally, (5) the last detects the data invariants. These critical invariants can uniquely define a given data structure.

3.2.2.1 Challenges of Detecting Data Structure Invariants

Detecting the invariants of data structures is very challenging. Unlike the original work of dynamic invariant detection which relies on the source code [49], MIDAS cannot simply record those target variables at function boundaries, because compilers can optimize an application thereby eliminating the function boundaries in the binary (inlining is one of the most basic optimizations triggered by compilers at the lowest level of optimizations). One might think of generating the traces when a load (store) instruction accesses the data structure—however, this does not help to solve the more serious problem of the ever-changing behavior of data structures which makes it even more difficult to detect their critical invariants.

Data structures change during runtime and temporarily lose their defined shape thereby invalidating their critical invariants. There are two reasons for this. On the one hand, a data structure can look like another instead of its defined shape, e.g, a simple (non-balancing) binary search tree can look like a list according to the particular data insertion and deletion patterns. In such a case, this work says that data structures suffer from *macroscopic change*, and MIDAS should report the data structure as a binary search tree other than a list. On the other hand, the defined shapes of data structures are temporarily corrupted due to disconnections or cycles created during *destructive updates* [117, 113]. For example, insertion into (removal from) the middle of a doubly-linked list makes it not look like a doubly-linked list. In this case, this work says that data structures suffer from *microscopic change*, and MIDAS should be robust enough to compensate for such changes and make sure it detects the doubly-linked list structure. To overcome these challenges, this work first define *dangerous traces* and then relies on an axiom to avoid them.

Definition 1 *Dangerous traces are defined as any traces that invalidate the shape or data invariants of a data structure.*

The dangerous traces are problematic, since the critical invariants are the basis for characterizing data structures. The followig axiom provides the insight to this problem.

Axiom 1 *When data structures lose the defined shape, their critical invariants do not hold.*

This Axiom translates to the fact that those traces generated while a data structure has lost its defined shape are *dangerous traces*. In other words, if data structures retain the defined shape, their critical invariants should hold. With that in mind, MIDAS figures out the defined shape of a data structure to filter out those traces generated while it loses the shape. The challenge is how to recognize the defined shape in the presence of constant changes of the data structre shape due to the macroscopic/Microscopic changes.

To achive this, MIDAS relies on the observation that for most of the time, a data structure shows the defined shape. That leads MIDAS to take an inductive manner to catch the shape; MIDAS associates each trace with *memory graph magnitude* which coarsely but effectively summarizes the ever-changing shape of data strucures over time. Then, MIDAS post-processes the collection of execution traces and filters out problematic traces which possess rare *memory graph magnitudes*. This ensures that only essential traces generated while data structures retain the defined shape are used into the invariant detector for their identification. The next section describes the *memory graph magnitude* metric and Section 3.2.3 explains how MIDAS correctly filters out the *dangerous* traces.

3.2.2.2 Monitoring Data Structure Change with Memory Graph Magnitude

As mentioned before, data structures change in both macroscopic and microscopic scales in terms of their shapes, and therefore MIDAS must keep track of such changes to discriminate data structure's defined shape. More specifically, MIDAS should be able to characterize how the data structure appears at a certain point

during execution as well as differentiate the shapes at each point in the meantime. To achieve this, this work introduces *memory graph magnitude*, a simple metric that coarsely and effectively summarizes the ever-changing shape of data structures.

It is important to note that the purpose of this simple metric is not to directly detect the data structure shape but to focus on the representative traces for preserving the shape/data invariants. Recall that rare traces are likely to be *dangerous* since they are generated when data structures lose the defined shape.

In the following, this work defines the *memory graph magnitude* metric, and presents Theorem 2 to show the metric has well-defined range to be used as a *magnitude*. To prove Theorem 2, Theorem 1 is presented as a preparation step.

Definition 2 *For a memory graph G = (V, E) where E is a set of child edges, a memory graph magnitude M(G), is a function M: $\mathcal{G} \to \mathbb{R}$ defined as*

$$M(G) = \frac{\|E\|}{\|V\|}\Delta(G) \tag{1}$$

where $\Delta(G)$ is a *maximum degree* [2] of a memory graph G.

Theorem 1 *Any two k-regular graph G_1 and G_2 have the same memory graph magnitude*

$$M(G_1) = M(G_2) \tag{2}$$

For any *k-regular graph G*, $\frac{\|E\|}{\|V\|} = k, \Delta(G) = k,$

$$M(G) = \frac{\|E\|}{\|V\|}\Delta(G) = kk = k^2 \tag{3}$$

This implies that *M(G)* is independent on $\|E\|, \|V\|$. Therefore, Every *k-regular graph G* has the same magnitude.

Theorem 2 *For a memory graph G = (V, E) with the maximum degree Δ(G) = k, M(G) takes the maximum value when G is k-regular graph.*

[2] *This paper uses the term degree and out-degree interchangeably according to the definition of the memory graph.*

We use proof by contradiction. Suppose the claim is false; that is, *M(G)* is not a maximum;

$$\exists G' = (V', E') \ s.t. \ M(G') > M(G) \tag{4}$$

Note that *G'* is not a *k-regular graph*.

$$\frac{\|E'\|}{\|V'\|} > \frac{\|E\|}{\|V\|} \tag{5}$$

because $\Delta(G') = \Delta(G) = k$. Then,

$$\frac{\|E'\|}{\|V'\|} > \frac{\|E\|}{\|V\|} = k \tag{6}$$

because *G* is *k-regular*. However, for $G' = (V', E')$, $\Delta(G')$ is *k*; that is, for every vertex $v \in V'$, $deg^+(v) \leq k$. Therefore, the total number of edges $\|E'\|$

$$\|E'\| = \sum_{v \in V'} deg^+(v) \ \leq \ \sum_{v \in V'} k \ \leq \ k\|V'\| \tag{7}$$

i.e., $\|E'\| \leq k\|V'\|$. Therefore,

$$\frac{\|E'\|}{\|V'\|} \leq k \tag{8}$$

This is a contradiction from (6). Finally,

$$M(G') = \frac{\|E'\|}{\|V'\|}\Delta(G') \leq kk = k^2 = M(G) \tag{9}$$

$$M(G') \leq M(G) \tag{10}$$

This is another contradiction from (4). Therefore, the Theorem 2 must be true.

Note that Theorem 2 holds for any *G = (V, E)*, no matter how big $\|E\|$ and $\|V\|$ are (see the Theorem 1). Intuitively, for a given memory graph *G = (V, E)* with the *maximum degree* $\Delta(G)$, the *memory graph magnitude M(G)* represents how the graph is close to a *k-regular graph*. Figure 7 shows how *memory graph magnitude*

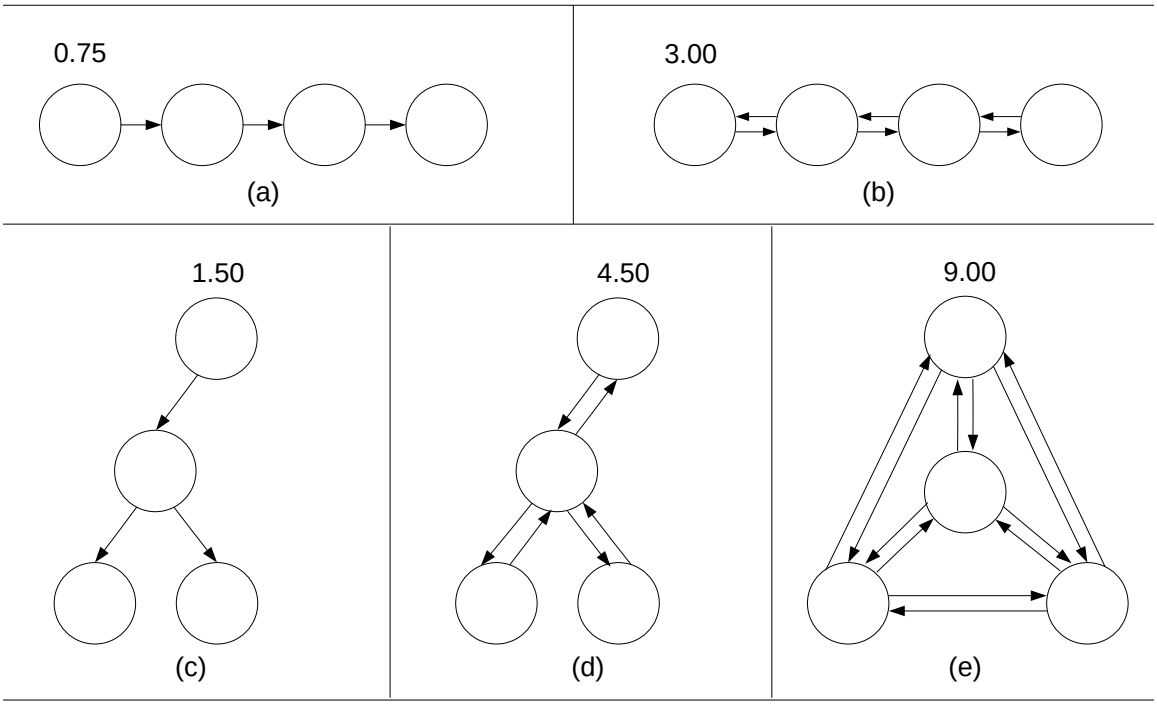


Figure 7: Examples of the *memory graph magnitude* with different types of graphs

is used to differentiate memory graphs with the same number of memory nodes. Each memory graph has its own *M(G)* , shown in the top of the graph, based on how tightly memory nodes are connected to each other and its *maximum degree*. As shown in Figure 7(a) and (b), it is said that a singly-linked list is less than a doubly-linked list in terms of *M(G)*. That is, the tighter connection between nodes leads to the greater *M(G)* . The same thing is with the memory graphs in Figure 7(c) and (d). Note that the memory graph in Figure 7(d) has the maximum degree of 3. In particular, a memory graph in Figure 7(e) has the maximum value, since it is a 3-*regular graph*.

One good side effect is that it becomes possible to infer what type of data structure is used just by evaluating the *memory graph magnitude M(G)* (even if our data

Table 2: The possible range of the *memory graph magnitude* for different data structures

Type (minimum $\|V\|$)	$\|V\|$	$\|E\|$	$\Delta(G)$	Possible range of *M(G)*
Singly-linked List (2)	n	n-1	1	$0.5 \leq \frac{n-1}{n}1 < 1$
Binary Tree w/o a parent (3)	n	n-1	2	$1.33 \leq \frac{n-1}{n}2 < 2$
Doubly-linked List (2)	n	2(n-1)	2	$2.0 \leq \frac{2(n-1)}{n}2 < 4$
Binary Tree w/ a parent (4)	n	2(n-1)	3	$4.5 \leq \frac{2(n-1)}{n}3 < 6$

structure detection currently relies on matching invariants with a predefined library of known data structures). I.e., it can serve as a clue to data structure identification in those situations where the invariant detection is too expensive or unavailable. It can also easily keep track of data structure evolution, e.g., when a data structure changes to which data structure. Lastly but not least, it can be used as a guideline for optimistic parallelization by comparing its value to the possible maximum magnitude based on theorem 2. For example, if *M(G)* value of the graph-like data structure is very close to the maximum magnitude, it should be better not to parallelize the data structure execution due to too many join points in it.

Table 2 shows the possible ranges of *M(G)* for some type of data structure comprised of n. Especially, since the first term of *M(G)*, i.e., $\frac{\|E\|}{\|V\|}$, is an increasing sequence, the *M(G)* has the minimum value when n is the minimum. As shown in the table, each data structure has its own *M(G)* range that is not overlapped with the range of others. Thus, matching a *M(G)* value with one of the ranges makes it possible to infer the type of the data structure.

Basically, the *memory graph magnitude* can strictly tell the order between data structures without a parent pointer. The same goes for data structures with a parent pointer. But, there can be an overlap between two *M(G)* ranges from data structures with/without a parent pointer, e.g., the *M(G)* of a quad tree without a

parent pointer can overlap with the *M(G)* of a doubly-linked list[3]. However, this is not a problem, since there is almost no possibility that data structures are evolving between the doubly-linked list and the quad tree. Even if there is such a case, MIDAS can effectively recognize the fundamental difference of data structures with the help of edge classification information in a memory graph. It must be noted that the *memory graph magnitude* can correctly recognize the case where data structures grow by adding a parent pointer, e.g., change from the binary tree without a parent pointer to binary tree with a parent pointer. In other words, it is guaranteed that the ranges of the two *M(G)* ranges of data structures with and without a parent pointer are disjoint, e.g., there is no overlap between *M(G)* values of the singly linked and the doubly-linked list. This is important in that some data structures evolve in a way that gets the connection between nodes in the memory graph tight by adding an new edge between them, e.g., a singly-linked list changes to a a doubly-linked list by adding an backward edge between the nodes. Section 3.2.3 shows such an example.

Note that unlike previous work [27], the *memory graph magnitude* metric considers only out-degree, not in-degree. The main reason is that in-degree is more vulnerable to transient behavior occurring during data structure manipulation, i.e., destructive updates. Another reason is that ignoring in-degree takes away the need for special care for connections between a data structure node and a sentinel node. It also needs to be noted that the metric is calculated on the fly by tracking a connected component of memory graphs and simply counting its numbers of memory nodes and child edges. Again, the metric is computed per data structure instance. The next section describes how MIDAS correctly filters out the dangerous traces with the help of the *memory graph magnitude*.

[3]In fact, it is possible to make *M(G)* completely differentiate data structures without any overlap. This can be done by rescaling the term, $\Delta(G)$ in its equation, e.g., $(\Delta(G))^3$ or $(\Delta(G))^4$, but we have not felt such a necessity.

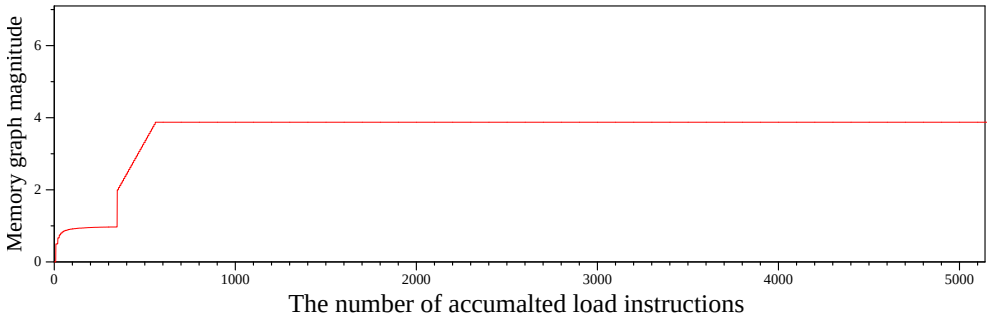
3.2.3 Detecting Data Structure Invariants in the presence of dangerous traces

MIDAS relies on the Axiom 1 to work around *dangerous traces* (See Section 3.2.2.1). With that in mind, MIDAS needs to recognize the defined shape of a data structure to filter out the dangerous traces. As a basis for recognizing the defined shape, this paper claims that data structures show their representative behavior thus showing their defined shape for most of the time, even if they may suffer the macroscopic and microscopic changes. E.g., for a tree, the duration during which it does not look like a tree is likely to be very short. And *destructive updates* are not representative behavior of data structures, either.

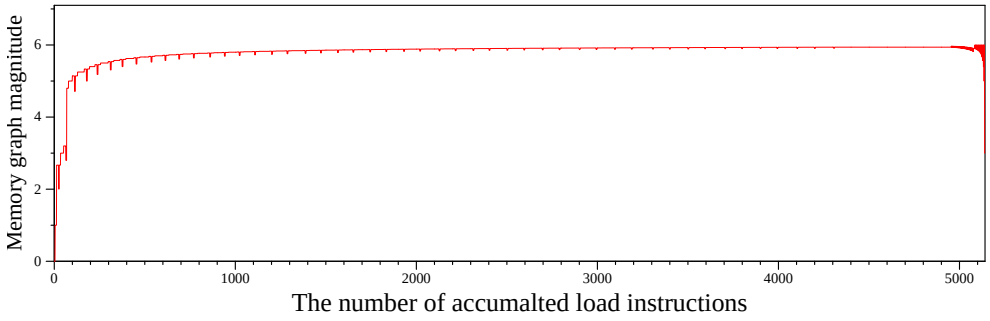
A key observation of *destructive updates* is that they are caused mostly by executing a few store instructions. With that in mind, MIDAS generates the traces of the target variables of the shape and data invariants along with *memory graph magnitude*, when a load instruction is executed instead of a store. This allows MIDAS to focus more on representative behaviors of data structures.

Before discussing how to filter out dangerous traces, this work first verifies the claim that the macroscopic and microscopic changes of a data structure, which easily invalidate its critical invariants, are not representative behaviors of a data structure which easily. Figure 8a and Figure 8b describes how MIDAS can capture both macroscopic and microscopic changes in the shape of data structures. In the figures, X-axis represents the number of executed load instructions while Y-axis represents the *memory graph magnitude* at a specific time that a load instruction is executed.

Figure 8a describes how the critical data structure of *Bh* from Olden benchmark suite changes as time goes by. The main data structure of *Bh* is a doubly-linked list, but at the beginning of the execution, the nodes of the list are connected by using only a next pointer in each node. I.e., the benchmark looks like a singly-linked list initially. Then, at some point, the benchmark traverses the list adding a new



(a) Macroscopic change of a data structure in Bh benchmark.



(b) Microscopic change of a data structure in STL `set` microbenchmark.

Figure 8: Data structure's (a) macroscopic and (b) microscopic changes; the memory graph magnitude of each data structure appears on Y axis.

back edge to each node, thereby changing the data structure into a doubly-linked list. As Figure 8a describes, for most of the time, the data structure exists in a state that *memory graph magnitude* represents its defined shape, a doubly-linked list. The implication is that in spite of such a macroscopic change in the shape, the data structure shows its representative behavior.

Figure 8b describes how STL `set` represents the microscopic changes in its shape. Note that `set` is implemented by using a red-black tree that has a parent pointer, i.e., the maximum degree is 3 in the memory graph. The microbenchmark simply inserts 100 integers in the increasing order. Therefore, the tree rotation is frequently occurred to re-balance the red-black tree. Such tree rotations are typical examples of *destructive updates*, and they are represented as a little spike of the curve in the Figure 8b. Overall, it turns out that microscopic changes due to the *destructive updates* are pretty sporadic.

Thus, these tests verify our claim that the macroscopic and microscopic changes are not representative behaviors of data structures, and therefore it may be possible to filter them out.

3.2.3.1 Filtering Out Dangerous Traces

It turns out that data structures show their defined shape for most of the time. Therefore, if traces are generated when a data structure keeps its defined shape, the traces should be dominant among the whole execution traces of the data structure. On the other hand, those traces should be rare that are generated when the data structure loses its defined shape due to the macroscopic and microscopic changes. This motivates MIDAS to focus on the common case in the entire traces of a data structure to recognize the defined shape of the data structure.

To achieve this, MIDAS post-processes the generated trace file at the end of program execution. The offline process finds out for each number of memory nodes which *memory graph magnitude M(G)* appeared the most frequently in the traces, i.e., the pairs of the *number of memory nodes* and the *mode M(G)* are discovered. MIDAS feeds only those traces that correspond to the pairs into the invariant detection tool. Thus, the tool can detect the invariants for the representative behaviors of a data structure which keep the shape and data invariants. In this way, MIDAS completely filters out dangerous traces due to the macroscopic (Figure 8a) and microscopic (Figure 8b) changes, since they rarely appear thus not belonging to the the pairs. Note that in the traces from first phase in Figure 8b are completely filtered out. This is very important to correctly detect the data structure, a doubly-linked list, since the data structure initially looks like a singly-linked list.

Sometimes, even the same data structure can show phase behaviors in terms of *memory graph magnitude*. That is based on the usage patterns of a data structure. Figure 9 shows such an example conceptually. During the first phase in the

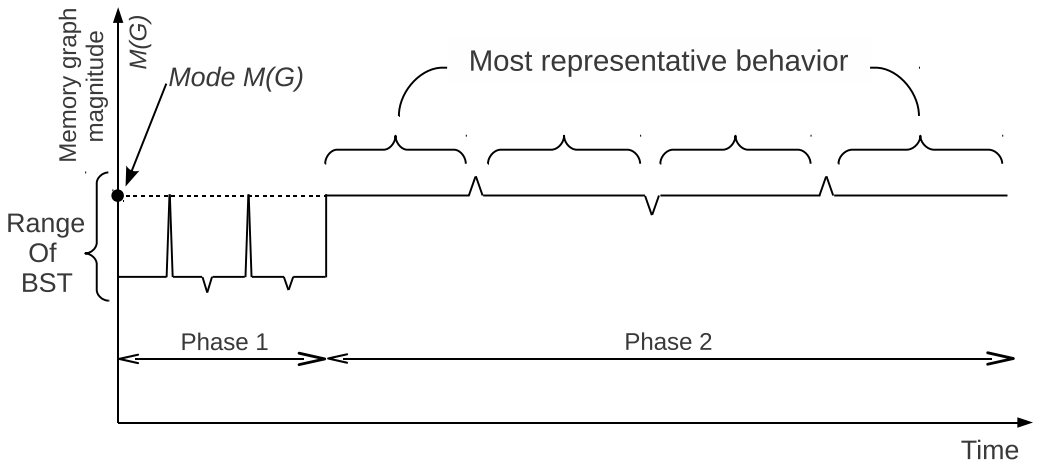


Figure 9: Phase change of a data structure. The spikes in the both phases represents abnormal behaviors of a data structure such as microscopic changes.

Figure, a binary search tree executes with some number of data elements. Even if the insertion/deletion/re-balancing operations are performed, the number of elements in the tree remains same as the initial number for most of the phase. In the second phase, the tree accommodates more data elements suddenly. And again in spite of tree operations, it mostly keeps the total number of data elements for the rest of the time. In this case, for the both phases, the data structure mostly shows its defined shape, a binary search tree. Especially, the data structure spends the most of its life time in the second phase, thus the first phase would not be the representative behavior. The lesson here is that data structure identification is still achievable with traces in the second phase. This leads MIDAS to perform the invariant detection only for the most representative behavior of data structures.

To accomplish this, MIDAS first finds out which *M(G)* appeared the most frequently, i.e., the same *mode M(G)* is computed. Note that many traces can have the *mode M(G)* as long as they have the same $\frac{\|E\|}{\|V\|}$ and $\Delta(G)$. With that in mind, MIDAS then finds out for the mode *M(G)* which *number of memory nodes* ($\|V\|$) appeared the most frequently, i.e., the pairs of the *mode M(G)* and the *mode* $\|V\|$ are discovered. Like tall spikes in Figure 9, those small spikes in the second phase are very likely to

be dangerous traces that have the *mode M(G)* but not the *mode* $\|V\|$. Again, MIDAS feeds only those traces that correspond to the pairs into the invariant detection tool. This strengthen MIDAS against the dangerous traces due to data structure's evolution (macroscopic change) and *destructive updates* (microscopic change).

These two schemes effectively filter out the dangerous traces thus they both identify data structures. Later in Section 3.3.2.3, these approaches are compared and discussed.

3.2.4 Matching Invariants

After performing the filtering scheme while preserving the critical invariants of data structures, MIDAS is ready to identify the data structures by matching the invariants. MIDAS relies on a library of pre-characterized data structures to compare against. This library contains a set of shape and data invariants for each candidate data structure. Against this library, a target data structure can be compared to determine what known data structure best matches the target, i.e., the presence of critical invariants is tested.

The invariants are picked that distinguish essential characteristics of each data structure, based on its definition rather than on implementation. That is, for a linked list, MIDAS attempts to look for an invariant, "*an old node is connected to a new node*" instead of "*a new node points to* `NULL` ". The latter is likely to be implementation specific. Intuitively, the shape graph invariants of a data structure determine how the data structure looks like, e.g., each node has two child edges. Meanwhile, the data invariants distinguish between those data structures which have similar shapes. Extending this library is an easy process: simply run a sample execution of an application with the target data structure, look through the list of identified invariants, and add the critical invariants into the library. In practice, a new data structure can be added to the library in a few minutes.

It is important to note that the library contain only *necessary* invariants. Thus if the dynamic instrumentation creates additional invariants that may be overly conservative, MIDAS does not require those conservative invariants to match what is in the library. Again, using invariant detection to categorize data structures is probabilistic in nature, and it is certainly possible to produce incorrect results. However, as Section 3.3 empirically demonstrates MIDAS can effectively detect different implementations from several real-world data structure libraries.

3.3 Evaluation

To evaluate the accuracy of MIDAS, we implemented it using PIN dynamic binary instrumentation tool [89]. The traces generated from the instrumented code are fed into Daikon [49] to detect invariants needed to identify data structures. These invariants are then compared with a library of data structures that was seeded with simple applications we wrote using the C++ Standard Template Library (STL) [125].

3.3.1 Demonstrating the Correctness of MIDAS

This section first demonstrates MIDAS' accuracy to identify encapsulated data structures using real-world data structure libraries.

3.3.1.1 Detecting Encapsulated Data Structures

Together with STL, we verified MIDAS' accuracy using two externally developed data structure libraries: the GNOME project's C-based GLib [132] and Borland C++ Builder's Standard Library STLport [127]. Several microbenchmarks were used to evaluate MIDAS' effectiveness across data structure implementations. These benchmarks were based on the standard container benchmark [12], a set of applications originally designed to test the relative speed of STL containers. These were ported to the various data structure libraries and run through MIDAS.

MIDAS correctly identified a set of recursive data structures from the three libraries. MIDAS reported that `set` in STL and STLport uses a red-black tree. To accomplish this, MIDAS successfully recognized the presence of the data invariants, e.g., the left child's data is less than the right child's data, and that each node contained a field which contains only two values: one for "red" and one for "black". Similarly, MIDAS reported that `GTree` in GLib uses an AVL tree. In particular, MIDAS recognized that each node has a field of the balance factor to strictly balance the AVL tree. MIDAS also detected that `list` in STL and STLport `GQueue` is implemented using a doubly-linked list. For `hash_map` and `GHashTable` from STL/STLport and GLib, MIDAS reported the data structures as an array of singly-linked lists.

In addition, we tested a non-balancing binary tree to evaluate MIDAS' accuracy against macroscopic changes of a data structure, varying data insertion patterns. MIDAS correctly identified the data structure in spite of many artificial insertion and deletion patterns that sporadically makes the data structure look like a singly-linked list and then recovers its defined shape soon.

On average, the overhead for instrumenting the code to recognize data structures was about 90X-120X. The instrumentation overhead was about 20X-30X while the invariant detection time comprises the rest of the overhead. The time spent on filtering traces is negligible. The memory space overhead is about 2.5-4x depending on how much heap memory is originaly used in an application. While this analysis does take a significant amount of time, it is reasonable to perform heavy-weight analysis like this during the software development process since it is once in a development cycle cost.

3.3.1.2 *Detecting Non-Encapsulated Data Structures*

To evaluate MIDAS' accuracy for non-encapsulated data structures, we use Olden benchmarks since their data structures do not have well-defined interface functions and some of them do not use the standard memory allocator. Thus, the benchmarks are appropriate to show how MIDAS works in the presence of a custom memory allocator. All the benchmarks were compiled with an option, "-O3" to show MIDAS' accuracy on optimized application binaries. Table 3 shows the main data structures of each benchmark applications and the reported data structures by MIDAS.

Table 3: The identification results of Olden data structures

Application	Main Data Structure	Reported Data Structure
Bh	doubly-linked list	same
Em3d	singly-linked list	same
Health	doubly-linked list	same
Mst	hash table	array of singly-linked lists
Perimeter	quad tree	quad tree with a parent pointer
Power	singly-linked list of singly-linked lists	same
Treeadd	binary tree	full binary tree
Tsp	binary tree two jump pointers	quad tree

For Bh, Em3d, and Health, MIDAS correctly reported their critical data structures. As shown in Section 3.2.3, Bh changes its data structure from a singly-linked list to a doubly-linked list. In this case, MIDAS successfully preserved the critical invariant that every two nodes are doubly linked to each other thereby reporting the data structure as a doubly-linked list. MIDAS correctly identified the main data structure of Perimeter which is a quad tree with a parent pointer. MIDAS detected the invariant that parent and child nodes are doubly connected in the tree. The main data structure of Power is a singly-linked list in which every node holds its own singly-linked list. MIDAS reported the data structure as a singly-linked

list of singly-linked lists. For Treeadd, MIDAS reported the data structure as a full binary tree. In particular, MIDAS recognized the data invariant that data values in the tree nodes have the same value, "1". Lastly, MIDAS reported the main data structure of Tsp as a quad tree. This is because MIDAS identified the two jump pointers in the tree node as child edges.

Overall, MIDAS successfully reported the main data structures of Olden benchmarks when they are compiled with aggressive compiler optimization. We also tested different compiler optimization levels, and found out that MIDAS consistently identifies the data structures. Thus MIDAS can accurately identify data structures irrespective of how encapsulated they are, and even of how optimized the binary is.

3.3.2 Analysis

This section first provides quantitative results to demonstrate MIDAS' abilities to tolerate *destructive updates*. We then verify whether MIDAS catches the representative behavior of data structures. Finally, we provide a way to reduce the overhead of the invariant detection tool.

3.3.2.1 How Robust is MIDAS against Destructive Updates?

It should be noted that even a single trace generated during *destructive updates* can invalidate the critical invariants of data structures. Therefore, MIDAS must be robust against *destructive updates*, i.e., it must completely filter out such a dangerous trace.

To evaluate how robust MIDAS is against *destructive updates*, two microbenchmarks, *list-set* and *tree-set*, were used that are two different *set* data structure implementations using a doubly-linked and a red-black tree, respectively. Basically, the both microbenchmarks insert a random number and erase another random number so that destruction updates are frequently performed inside a loop body as

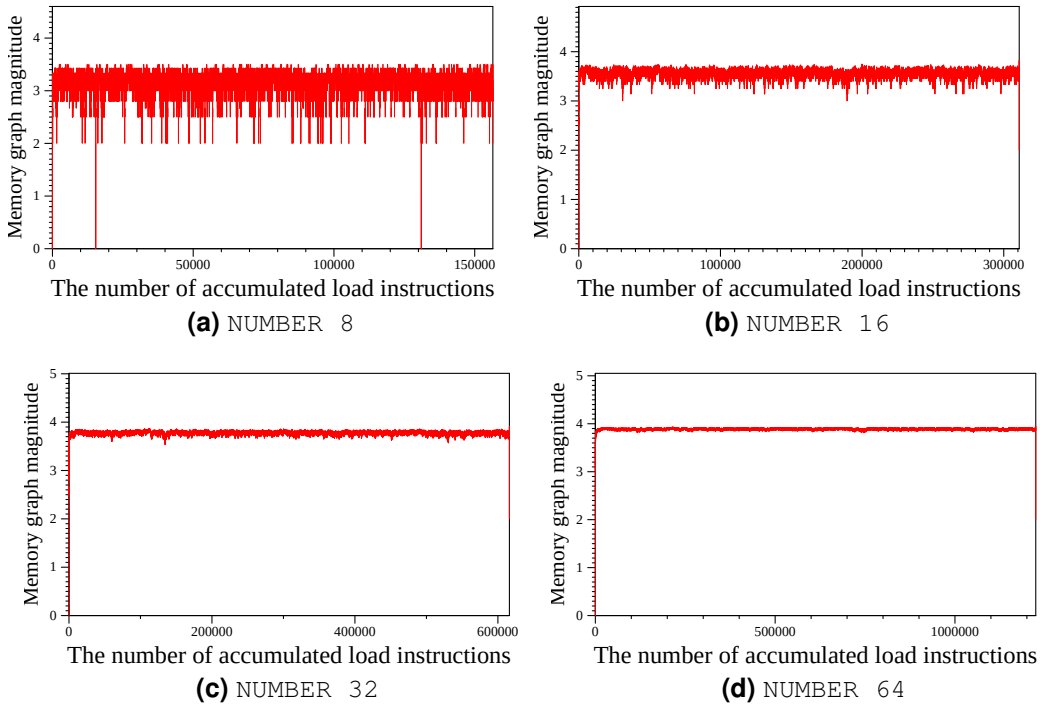


Figure 10: Memory graph magnitude for the *list-set* benchmark with varying NUMBER

follows;

```
set.insert(rand() % NUMBER)
set.erase(rand() % NUMBER)
```

In particular, *list-set* inserts a new data element to the random position to cause more *destructive updates*, i.e., data elements can be inserted to an arbitrary position in the list. Thus, the benchmarks cause *destructive updates* frequently according to the NUMBER on `insert` as well as on `erase`.

Figure 10 and Figure 11 describe how the microbenchmarks suffer from *destructive updates* when the NUMBER is varying. In the Figures, more fluctuation of the *memory graph magnitude* means that more *destructive updates* are occurring, thus the smaller NUMBER is, the more *destructive updates* are. In particular, *tree-set* suffers from more severe *destructive updates* than *list-set* does for the same NUMBER. That

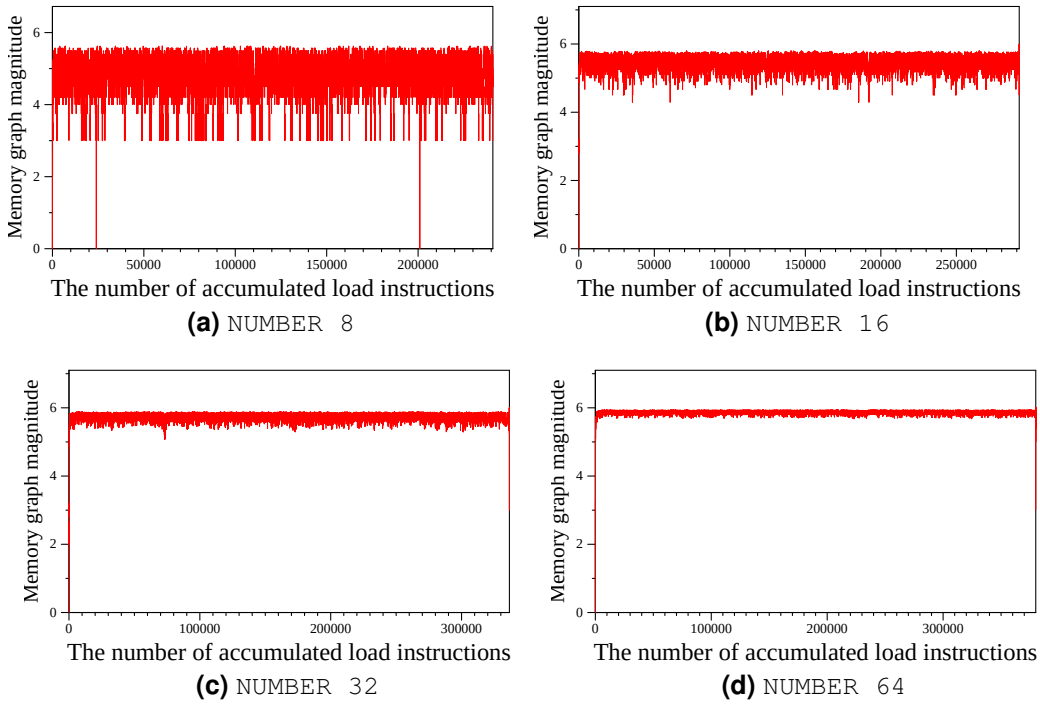


Figure 11: Memory graph magnitude for the *tree-set* benchmark with varying `NUMBER`

is because tree rotations cause *destructive updates* not only in `erase` operations but also in `insert` operations.

For each benchmark configuration, we investigated whether critical invariants of the data structure hold. It turns out that for the benchmarks, the critical invariants of the data structures always hold. We show some invariants of the *tree-set* in Daikon output syntax [49].

```
child_edges_per_memory_node of { 0, 1, 2, 3 }
child_edges: - 2 * memory_nodes: + 2 == 0
node.data_offset_0 : one of {0, 1}
node.child_offset_8.data_offset_0: one of { 0, 1 }
node.child_offset_16.data_offset_0: one of { 0, 1 }
node.child_offset_24.data_offset_0: one of { 0, 1 }
node.data_offset_32: > ::node.child_offset_16.data_offset_32:
```

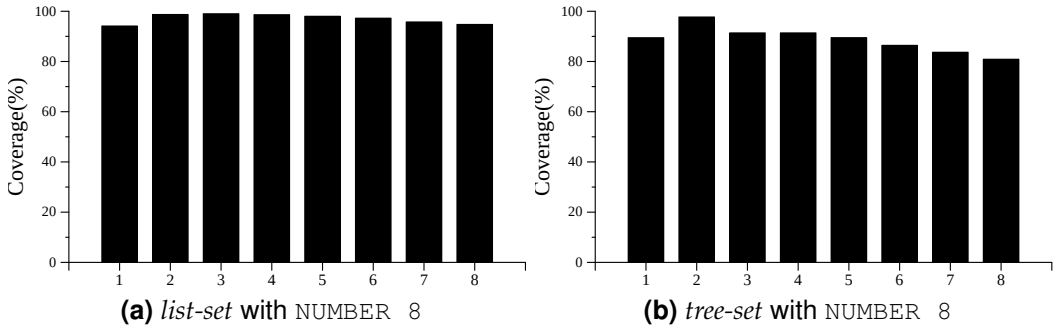
`node.data_offset_32: < ::node.child_offset_24.data_offset_32:`

The first invariant tells that each node has at most 3 connections with another. Note that the red-black tree in *tree-set* has a parent pointer. The invariant, "*child_edges - 2 * memory_nodes + 2 == 0*", tells us that every two nodes are doubly linked to each other. The next four invariants represent that there is a data value that always holds one or zero, the color of a red-black tree node. The rest invariants represent that the value in a memory node is always larger than the first child and smaller than the other child. Consequently, MIDAS can correctly identify the data structures against excessive *destructive updates*.

3.3.2.2 Does MIDAS capture representative behavior of Data Structure ?

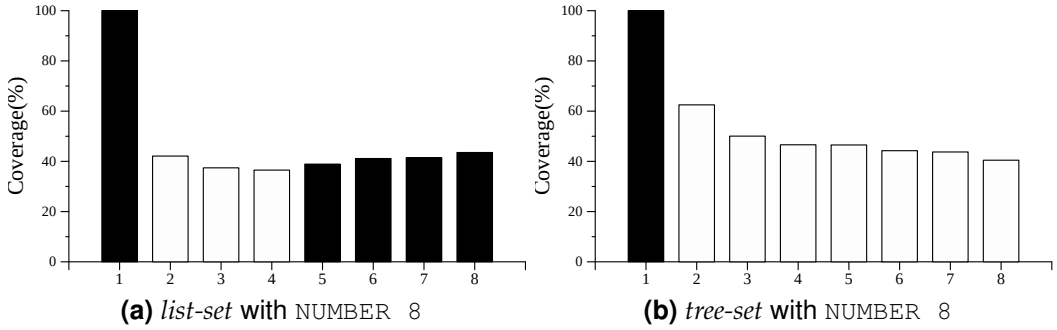
As shown in previous section, MIDAS can detect accurately data structures in the presence of severe destructive updates by filtering them out. The achieve this, MIDAS focuses on the common case in the entire traces on the assumption that *destructive updates* are relatively rare (See Section 3.2.2.1. In other words, it is assumed that those traces selected by MIDAS for invariant detection should be the representative behavior of data structures. This section verifies this assumption in case one might suspect that MIDAS would cherry-pick a small number of traces to make the critical invariants hold; if MIDAS would not focus on the representative behavior, the identification result would be less convincing. With that it mind, we leverage the term *coverage*, which is the fraction of entire execution traces that are selected and fed into the invariant detection tool to identify the data structure. Intuitively, *coverage* represents how dominant the selected traces are.

Figure 12 shows the coverage of the both microbenchmarks when the `NUMBER` is 8, which is the case of the most severe *destructive updates*. In the Figure, the *coverage* is represented for each number of data elements. Note that in this case, the microbenchmarks can have at most 8 elements. For example, the fifth bar shows



(a) *list-set* with `NUMBER 8` **(b)** *tree-set* with `NUMBER 8`

Figure 12: Coverage of `load` time generated trace



(a) *list-set* with `NUMBER 8` **(b)** *tree-set* with `NUMBER 8`

Figure 13: Coverage of `store` time generated trace

the *coverage* of the traces generated when the data structure contains five data elements. For each number of elements in *list-set*, the *coverage* is almost close to 100%. In *tree-set*, even if it suffers from more *destructive updates*, every *coverage* is still high, which means the selected traces for invariant detection are dominant. This confirms that those traces, which make the critical invariants of a data structure hold, are dominant, i.e., they reflect the representative behavior of the data structure. In addition, this shows that MIDAS verifies almost all accesses to data structures with the invariant detection. Thus the identification results are credible, and it is expected that MIDAS can be leveraged for verifying the properties of data structures in spite of its probabilistic nature.

It should be noted that such high *coverage* mainly results from the `load` time trace generation. The implication is that load instructions are rarely involved in data structure's abnormal behaviors. Recall that *destructive updates* are realized

with a few consecutive store instructions. What if MIDAS uses `store` time trace generation? Figure 13 shows this situation with the same configuration of the both microbenchmarks. In the both microbenchmarks, most of *coverage*s are below 50%. This shows the traces selected for invariant detection are not dominant, even though they appear the most frequently in the whole trace. The real problem is that among the selected traces, there are ones generated during *destructive updates* shown in empty bars in Figure 13. Thus, `store` time trace generation cannot provide MIDAS with a way to recognize the representative behaviors of data structures. In that case, MIDAS cannot report data structures correctly.

3.3.2.3 Optimization to Reduce Traces to Invariant Detection Tool

One problem of the invariant detection tool is that it is very slow and requires a lot of memory. This can be a problem since the invariant detection tool sometimes requires more than a GB memory, thus for low-end machines with small memory, MIDAS might end up with out-of-memory error. In general, the more traces are fed into the tool, the more memory are required and the slower the analysis time is. Therefore, reducing the number of traces fed into the tool directly address this problem.

MIDAS attacks this by taking both online and offline approaches. Online approach is to reduce the number of traces generated at runtime. MIDAS generates traces only when `load` instructions access the *data* part of a data structure [4]. I.e., no trace is generated when the recursive backbone of data structures is accessed. This allows MIDAS to avoid generating traces for the `load` instruction generated while traversing data structures without touching data element itself. One good side effect of this approach is that MIDAS can avoid unnecessary traces generated during *destructive updates*.

[4] This is achieved with the help of the edge typing in the memory graph of a data structures. See Section 3.2.1.

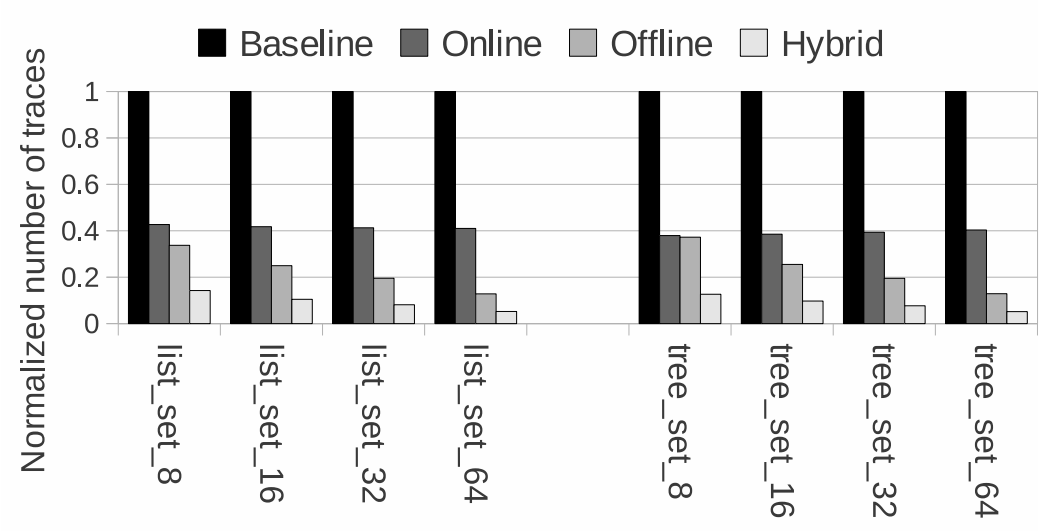


Figure 14: Normalized number of traces

The offline approach is to reduce the number of the generated traces fed into the invariant detection tool. The basic idea is to select only the traces that correspond to the most representative behaviors of data structures in terms of *memory graph magnitude M(G)*. Again, those rare traces that accidently have the *mode M(G)* are filtered out (See Figure 9). Figure 14 shows how the number of those traces that are processed for checking invariants varies for three cases; Online, Offline, Hybrid: combination of the previous two approaches. This experiment uses the same benchmark configuration as in Section 5.3.6.

Overall, the online approach reduces the traces of Baseline (no optimization case) by almost 60% on average. This results from the fact that once data elements are inserted into a data structure, they are searched and traversed a lot over the data structure. The offline approach reduces the original trace entities by 75% on average. Finally, the hybrid approach, online + offline, works the best creating synergy, i.e., it reduces the original trace entities by almost 90% on average. In spite of reduced traces, the results of data structure identification are consistently correct for all the benchmarks.

3.4 Summary

Identifying what data structures are used within an application is a critical step toward application understanding and many other aspects of program optimization. This work presents MIDAS, a framework for mining data structures from an application binary. MIDAS is a fully automated approach with no user intervention. During program execution, MIDAS traces the shape and data invariants of a data structure. These invariants can uniquely define the data structure. In particular, MIDAS automatically filters out those traces generated while a data structure loses its defined shape, thus preserving the critical invariants of the data structure. This paper demonstrates that MIDAS is highly accurate across several different implementations of standard data structures and non-encapsulated data structures of which the binary was optimized.

CHAPTER IV

DATA STRUCTURE SELECTION

4.1 Introduction

Niklaus Wirth famously noted, "Algorithms + Data Structures = Programs" [142], and it follows that one of the most critical aspects of creating effective applications is data structure selection. Data organization is one of the defining characteristics in determining how effectively applications can leverage hardware resources such as memory and parallelism. Indeed, it is not uncommon to find situations where simply changing the data structures can result in orders of magnitude improvement in application performance for many important domains. For example, scientific applications leveraging matrix inversion [30] and matrix multiplication [141], information mining from large databases [7], and analyzing genetic data for patterns [54], are instances of criticality of data structure selection in an application tuning process. According to [30], proper data structure selection can make the 2-D table implementation used in that study 20 times faster.

In one recent study, researchers at Google analyzed the use of the C++ Standard Template Library (STL) [125] on several of their internal applications, and found many instances where expert developers made suboptimal decisions on which data structures to use [87]. Simply changing a single data structure in one application resulted in a 17% speedup in that study. When applying this type of speedup to data-center-sized computations, poor data structure selection can result in millions of dollars in unnecessary costs. Thus, selecting the appropriate data structures in applications is an important problem.

However, the reality is that most often developers do not select data structure

implementations at all; they simply rely on a data structure library and assume that the library designer made a good decision for them. Data structure libraries were designed to be effective in the common case, and often leave considerable room for improvement in application-specific scenarios.

When developers do manually select a data structure implementation, they most frequently utilize asymptotic analysis to guide their decision. Asymptotic analysis is an excellent mathematical tool for understanding data structure properties; however, it often leads to incorrect conclusions in real systems. For example, comparing the STL `set` (implemented as a red-black tree) with `unordered_set` (implemented as a hash table), the `set` has worse asymptotic behavior but almost always has faster lookup times on modern architectures when holding fewer than 200 data elements. In other situations data structures have identical asymptotic behavior but very different real-world behavior. For example, splay trees [122] almost always perform better than red-black trees on real-world data though they have the same asymptotic complexity. Asymptotic complexity measures were designed as a unified basis for comparing and choosing an algorithm and not data structures. To a large extent, once an algorithm is chosen, attention is rarely paid to the choice of data structures. This can leave substantial inefficiencies on the table. In short, traditional solutions leave much to be desired.

Unfortunately, selecting the best data structure for a given situation is a very difficult problem. This requires thorough understanding both of how a program uses a data structure, and of the underlying architecture. Even further, input changes can lead to different optimal data structures. Thus, a tool that ignores inputs could not possibly make a high-quality decision for selecting the best data structure. To ameliorate the data structure selection problem, this paper presents Brainy, an automated tool to develop a repeatable process for creating accurate

cost models that predict the best data structure implementation for a given application/input/architecture combination.

In order to construct an input- and architecture-aware cost model, the model must be trained to understand the effect of architectural behaviors while taking into account input changes. This is accomplished by first constructing a set of synthetic programs that exercise different behaviors of a given data structure under consideration. For example, the test programs will stress all of the data structures' interface functions with modeling different inputs by varying data type sizes and various numbers of elements stored in the data structure. Then several measurements are collected through hardware performance counters and code instrumentation in order to understand how each data structure behaves. These measurements are then summarized into statistics which are then fed into a machine learning model. The machine learning model creates a function to accurately determine the optimal data structure choice for each static program variable. Machine learning characterization has been shown repeatedly to be more effective than human designed models because machine learning picks up on subtle interactions human experts often miss [46, 79, 99, 126, 137]. This paper demonstrates that leveraging machine learning to generate cost models, which leverage architectural events and dynamic software behavior, is significantly more accurate than asymptotic analysis or human designed models for data structure selection. This paper also demonstrates that using these models can result in significant performance improvements in real-world applications. Moreover such techniques are shown to be repeatable empirically on two different architectures across a variety of data structures.

The vision of this work is that the synthetic program generation tool we have developed can be used to tune a cost model once for each target system at install-time. These models can then be used either by a developer manually (e.g., as part of a performance debugging tool similar to Intel's VTune), or built into data

structure libraries so that the compiler or runtime can automatically select the best implementations for many users of the libraries. Utilizing machine learning to automatically generate cost models for data structure selection is a fundamentally new way to analyze data structure behavior; this method is significantly more effective than the traditional asymptotic analysis.

The contributions of this work include:

- A repeatable methodology for characterizing the performance of data structures using architectural events and runtime software properties of the application.
- An analysis on what program and hardware properties are most important to consider when selecting data structure implementations on modern architectures. This paper presents several non-intuitive discoveries. For example, branch misprediction rate is a very useful predictive feature.
- An empirical demonstration of the machine learning model, compared with traditional hand-constructed and asymptotic methods. This paper demonstrates that considering performance counters and dynamic properties can provide significant improvements in application performance.

4.2 Motivation

Effective data structure selection requires thorough understanding of how a data structure interacts with the application. Apart from the asymptotic behavior of data structures, a number of factors should be considered, such as what types of functions interact with the data, how many times the interface functions are invoked, how big each data element is, and so on. It is also important to take into account hardware behavior to understand the effect of the underlying architecture on data structure related code. Given all this, identifying a function that accurately

predicts the best data structure implementation is very challenging.

As an example, assume that a developer is deciding between a `vector` and `list` data structure from the C++ Standard Template Library (STL) [125]. The former is a dynamically-sized array stored contiguously in memory and the latter is a doubly-linked list. The developer might think that `vector` is almost always better than `list` because its contiguous data layout better leverages spatial locality in memory hierarchies, and the dynamically adjusting size will make tail insertions require fewer memory allocations than with a linked list. In reality, `vector` is preferable in situations with frequent search or iteration over data elements. However, data insertion into (or removal from) the middle of the structure is extremely expensive for `vector`, since all data elements located after the insertion point must be moved backwards (or forwards) to maintain contiguity. The challenging issue is how to quantify the pros and cons of each data structure to accurately compare them. For example, how many `find` or iteration operations are enough to overcome poor insertion and deletion times for `vector` to perform better than `list`? In some sense, we are looking at performing amortized analysis of different operations that are associated with a given data structure. Purely basing such an analysis on the frequency of operations would be a naive simplification of the problem, since the operations and their costs are linked to the program state and are continuously varying throughout the execution. It is a challenge about how to come up with such an amortized cost model without worrying about the deeper notions of the program state; a challenge partially solved by this paper. We first delve on this issue of generating an appropriate cost model.

Without worrying the issues of program state, one could limit oneself to the interface functions and their order of executions, and try to approximate the model of behaviors exercised. In general, constructing a cost function is much more difficult than illustrated by the above example, since there are many functions that

interact with each data structure. The best data structure implementation changes as each interface function is invoked more or less frequently relative to the others. Beyond just interface functions, any changes in data element size, the number of data elements, data search pattern, and so on, which can be affected by program inputs, can have a significant impact on the most appropriate data structure implementation. For example, STL's `find` searches for the first instance of a data element located in the structure without iterating over all the elements. This means the data being stored affects how important iteration is to the performance of the application. These and other input-dependent factors make it very difficult to hand-construct accurate data structure cost models.

A final challenge is that underlying hardware can have a considerable effect on data structure selection results. Even if a programmer chooses the best data structure, that data structure will not always be the best when it runs on different microarchitectures. That is, architectural changes can make the data structure, which was the best, suboptimal as input changes. For instance, in the previous example of data structure selection, a developer might choose a `vector` over a `list` for fewer cache misses during iteration, although hardware systems with larger cache sizes might execute `list` faster than `vector`. The reason is that `list` nodes will typically remain cached after a cold start; however, whenever `vector` is resized the cold start penalty will have to be paid anew. Thus, architectural events have a very important role in data structure selection.

To further motivate the importance of microarchitectural differences for effective data structure selection, this work analyzed several thousand randomly generated applications that exercise different behaviors of C++ STL data structures (further details on the application generator will be discussed in Section 4.4.1). Each application was run on both an Intel Core2 Q6600 and an Intel Atom N270 to see what the best data structure implementation for each architecture is. Figure 15

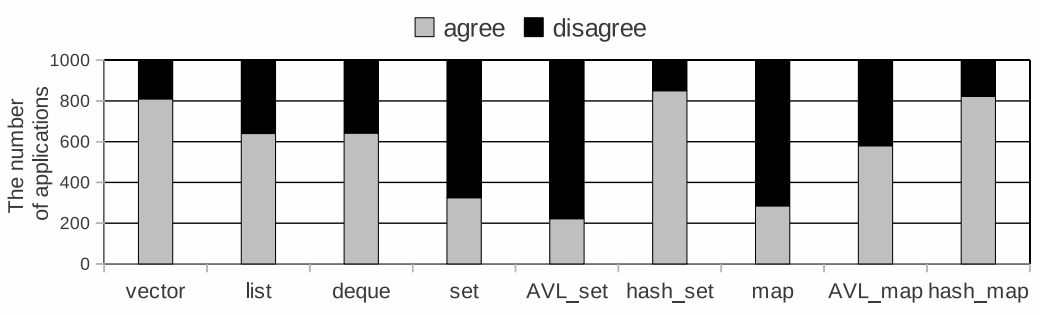


Figure 15: Different data structure selection results on two microarchitectures: Intel Core2 Q6600 and Intel Atom N270. Each bar represents 1000 applications whose best data structure on the Core2 is shown in the x-axis. For each application, if the data structure remains the same on the Atom, the application is classified as "agree". Otherwise, the application is classified as "disagree".

shows how differently two distinct microarchitectures can behave. Each bar in the figure represents 1000 randomly-generated applications whose best data structure implementation on the Core2 is shown on the x-axis. For example, the left-most bar in the figure represents 1000 applications whose best data structure on the Core2 was a `vector`. The dark gray, top portion of the bar represents how many of those exact same applications the best data structure was not a `vector` on the Atom architecture. So in ≈200 applications where `vector` performed best on the Core2, another data structure would perform better on the Atom.

This experiment demonstrates that the best data structure choice for each application significantly differs on the two different microarchitectures. The degree of such an inconsistency varies across data structures. On average, 43% of the randomly generated applications have different optimal data structures. Thus, all efforts to construct a data structure cost model *without* considering architectural properties will necessarily be lacking. The complexity of modern architectures further motivates the need for an automated tool to construct these models, as human-constructed models will be tedious and likely inaccurate. Section **??** shows that it is inherently difficult and sometimes impossible for hand-constructed models to capture the architectural events of an alternative data structure. E.g., the

number of branch mispredictions in the original data structure has no causal relation to that in the alternative data structure.

4.3 Overview

The purpose of this work is to provide a tool that can report the best data structures for different situations due to specific input sets and underlying hardware architecture changes. To keep up with the various behaviors of an application, this work exploits dynamic profiling that utilizes runtime instrumentation. Every interface function of each data structure is instrumented to model how that data structure interacts with the application. The instrumentation code observes how the data structure is used by the application (i.e, software features), and at the same time monitors a set of performance counters (i.e., hardware features) from the underlying architecture. The runtime system maintains the trace information in a context-sensitive manner, i.e., the calling sequences are considered at the data structure's construction time. This helps developers know the location in the source code of the data structures to be replaced. Once program execution finishes, the trace files are fed into a machine learning tool. Finally, the machine learning tool reports what data structures should be replaced with which alternatives.

Due to a significant amount of effort involved, to train and build machine learning models for the data structures, this paper limits its focus to C++ programs using a subset of the STL. It may be noted that as the tool is not fundamentally limited, the approach should be applicable to other data structures expressed in other contexts. To determine the target data structure replacements, we surveyed programs using Google Code Search (GCS) [53]. GCS indexes many open-source projects on the Internet. Figure 16 shows the number of static references to each data structure type across the entire index. This figure shows that `vector`, `list`, `set`, and `map` are the most common STL data structures used, thus this paper

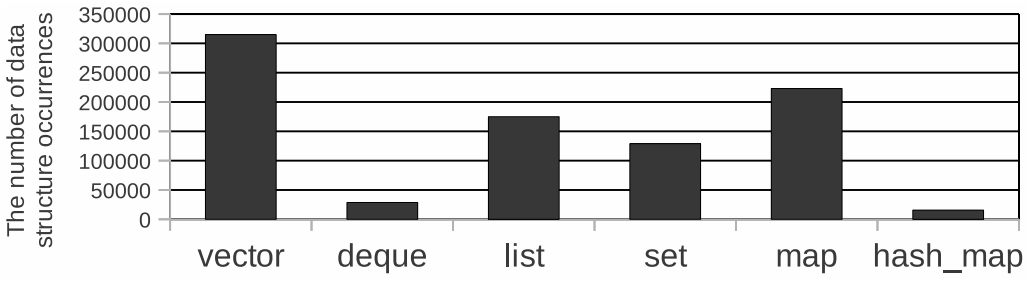


Figure 16: The number of data structure occurrences in all the code registered in Google Code Search.

will focus on various implementations of these structures. Simply counting the number of static references to each data structure ignores the importance of data structure's impact to the application performance at runtime. However, this gives a rough estimate for which data structure needs to be targeted initially.

Given this set of target data structures, it is also necessary to define a set of implementations, and delineate what implementations can be replaced by what. Table 4 shows the possible data structure replacements considered, along with the benefits and limitations of each. For example, `vector` can be replaced with `list` for faster insertion, and with `set` for faster search. Similarly, if `vector` is frequently searched with a key for a match, e.g., using *std::find_if*, then it can be replaced with `map`. However, `vector` cannot always be replaced by `set` or `map` because they are oblivious to the data insertion order (i.e., order-oblivious); Since they internally sort data elements, iteration over them leads to the sorted sequence of the elements. Therefore, iterating over the `vector` precludes these replacement candidates. Those particular implementations in Table 4 were chosen because they are already implemented within the STL, and other implementations could easily be added to the cost model construction system.

With this set of target implementations in mind, Figure 30 shows a high-level diagram of the proposed usage model. At compile time, an application is linked

Table 4: Data structure replacements considered for each target data structure.

DS	Alternate DS	Benefit	Limitation
vector	list	Fast insertion	None
	deque	Fast insertion	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast insertion & search	Order-oblivious
list	vector	Fast iteration	None
	deque	Fast iteration	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast search	Order-oblivious
set	avl_set	Fast search	None
	vector	Fast iteration	Order-oblivious
	list	Fast insertion & deletion	Order-oblivious
	hash_set	Fast insertion & search	Order-oblivious
map	avl_map	Fast search	None
	hash_map	Fast insertion & search	Order-oblivious

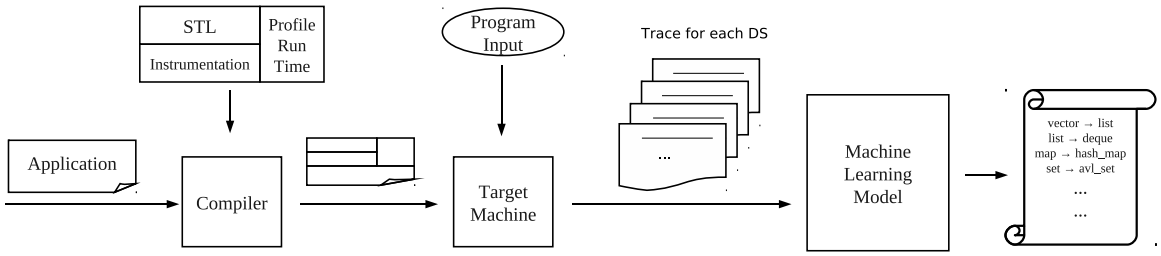


Figure 17: The framework of the data structure selection.

with a modified C++ Standard Template Library (STL) so that profiling data structures are used instead of the original ones. The profiling data structures are inherited from the original STL data structure, and their interface functions contain code which records the behaviors including hardware performance counters, and then calls the original interfaces. All the profiling features are recorded in trace files, which are post-processed and sorted by data structure. This sorting takes both relative execution time and calling context into consideration, in order to provide developers with a prioritized list of which data structures are most important to change. Once the data is sorted, the machine-learning-based cost model provides

a suggestion of what data structures should be replaced with alternate implementations. Optionally, this output could be fed into a code refactoring tool [92], which could automate the implementation replacements. This type of optimization tool can have a significant impact on the performance of real-world applications.

4.4 Model Construction

Accurate model construction is essential for effective data structure selection. Brainy leverages machine learning to construct the model for predicting the best data structure implementations. The model must satisfy three properties to be successful. First, the model should be accurate across many different data structure behaviors and usage patterns. Second, the model should be aware of microarchitectural characteristics of the underlying system. Third, the methodology for characterizing the performance of data structures should be automated and repeatable so that it is easy to construct new models for new microarchitectures.

If these properties are not satisfied by the model, architectural variations would easily make the predicting performance of the model inaccurate. In this case, improving the accuracy of the model requires re-training the model on the new microarchitecture. A more serious problem is that the training applications/examples[1] painfully-collected to cover the huge design space on the original microarchitecture might not provide abundant learning capabilities any longer on the new microarchitecture (See Figure 15). That is, due to the architectural change, the original training applications could not produce the broad spectrum of the best data structures as before, thus failing to model various data structure behaviors. Therefore, new training applications should be collected again to cover the missing portion of the design space. This is extremely time-consuming and requires enormous effort without the help of the automated and repeatable methodology. This section

[1]This paper uses the terms ”training applications” and ”training examples” interchangeably.

describes how these issues are addressed. It must be noted that just using machine learning itself cannot satisfy the issues. These issues are rather the prerequisites for the success of machine learning.

Formally, the description of the data structure selection model is as follows: given a set of input features X and a set of data structure implementations Y as output, the model is to find a function $f: X \rightarrow Y$ such that the predicted result $y = f(x)$, where $y \in Y$ and x is a set of features for a data structure in an application, matches the best data structure (*BestDS*) of the application. The training set of the model is comprised of many pairs of the feature set and the best data structure, i.e., $(x^1, BestDS^1)$, $(x^2, BestDS^2)$, ..., etc. The features include both software features such as the number of interface invocations and hardware features such as cache misses (Section 4.5.1 discusses the both features in more detail). Thus, features capture various aspects of the data structure usage when an application is running. In collecting the training set, Brainy uses an application generator to prepare a significant quantity of applications and executes each application through two phases of data collection: first to measure the runtime and second to record the detailed performance metrics. This section describes why so many applications are required, the details of the application generator, and how it is used in the two phases of data collection.

4.4.1 Training Set and Overfitting

Creating an accurate model using machine learning that represents a vast array of different data structure behaviors requires having a large and thorough set of training examples. If the training examples are not representative of the many varied behaviors of real world applications, then the resulting model cannot yield

accurate predictions. Therefore, training should provide the machine learning algorithm with all critical patterns of data structures' behaviors in which one implementation performs much better than another. Unfortunately, constructing such a training set is a very difficult problem.

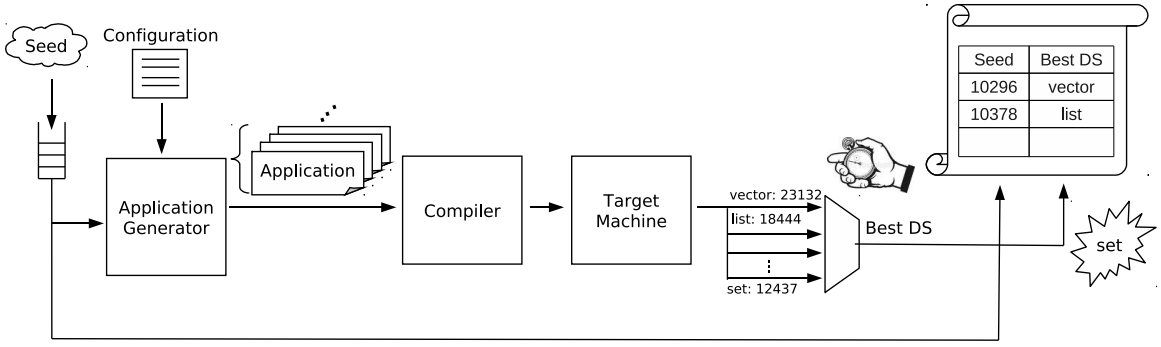


Figure 18: Training Framework Phase-I; Generating Applications and Measuring Execution Times

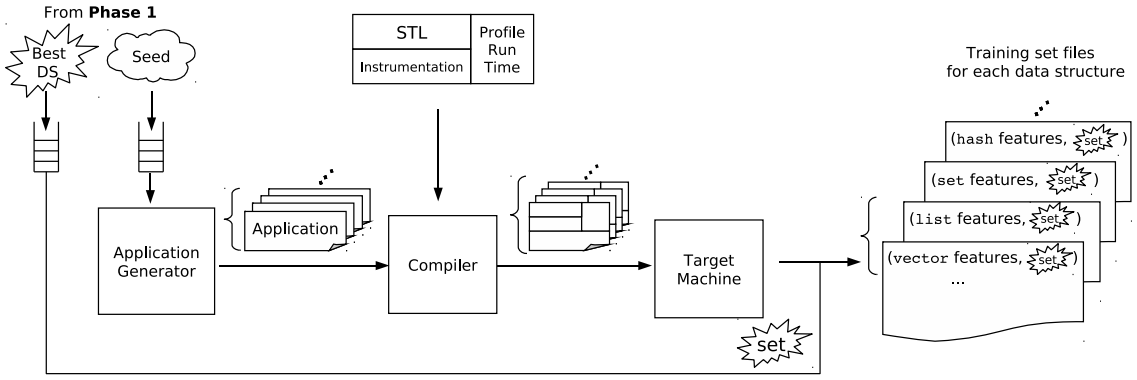


Figure 19: Training Framework Phase-II; Collecting Software and Hardware Features

The main difficulty of constructing effective training example sets is the very large design space. For example, an application may use only a subset of interface functions, or use them with a consistent frequency distribution (e.g., always performing twice as many lookups as insertions). On top of that, there are many hardware-specific characteristics, such as the size of data elements in relation to cache-block size, that make the training example sets constructed for one architecture potentially irrelevant for another.

Compounding the problem, each portion of the design space must be fully represented in order to avoid *overfitting* the model. *Overfitting* is a well-documented problem where machine learning algorithms adjust to random features (i.e., noise) of the training examples. Since such random features have no causal relation to the prediction function, the resulting prediction performance on unseen data becomes poorer while the performance on the training examples improves [42]. Thus, *overfitting* misleads the resulting model away from the optimum. This is most likely to become a severe problem for insufficient amount of training examples, since the noises are much more outstanding in that case, i.e., the model is inevitably inaccurate.

Because of the immense search space and the problems from *overfitting*, sample benchmarks cannot effectively train a machine learning model for data structure selection.

4.4.2 Application Generator

Instead, this work proposes using an *application generator* to cover the design space sufficiently with synthetic applications. That is, a tool (the application generator) creates a variety of applications that test different parts of the overall space. Each application models particular behaviors of a single data structure which are randomly determined, i.e., a probability distribution determines how the interface functions should be invoked. Using the application generator, Brainy can easily have as many training examples as needed, thereby avoiding the *overfitting*. Note that if there are a sufficient number of training examples, then the noise would play a vanishingly small role in the learning process. The vision is that the application generator and the configuration file can be distributed with the data structure library, and can be used to train the machine learning model at install-time for the specific hardware of the system.

Table 5: The behaviors of a data structure which are randomly decided, and the specification example in a configuration file.

DS behavior determined randomly	Specification example	Description
Total # of calls for all interface functions	$TotalInterfCalls = 1000$	Randomly divide 1000 calls and assign them to each interface
Size of data element	$DataElemSize = \{4, 8, 64, ...\}$	Randomly pick one from the specified set
Maximum value of data to be inserted	$MaxInsertVal = 65536$	Insert a random number between 0 and 65536 on `insert`
Maximum value of data to be removed	$MaxRemoveVal = 65536$	Remove a random number between 0 and 65536 on `erase`
Maximum value of data to be searched	$MaxSearchVal = 65536$	Search a random number between 0 and 65536 on `find`
Maximum # of data elements to be iterated	$MaxIterCount = 65536$	Iterate data elements a random # of times under 65536 on `++`/`--`

The application generator first prepares a synthetic application with an abstract data type (ADT) implemented by a C++ template that can take each data structure. The modeling is achieved via randomization. To illustrate, the synthetic application runs a function-dispatch loop. A random number determines which interface function is invoked every iteration of the loop. Thus, the order of interface invocations and their invocation frequencies are random. Randomization also controls how the dispatched interface is invoked, e.g., what data element is searched for `find`. Table 5 represents what property is randomly determined, and how it is specified in a configuration file. In particular, this configuration only specifies the total number of all the interface invocations. In each generated application, the number of invocations of each interface may vary, but the total number of invocations is constant across the applications.

To cover the different behaviors of interface invocations, the number of invocations for a given interface should be able to vary between zero and the total number. To achieve this goal, Brainy exploits a random number distribution to choose the number of invocations for each interface, such that the sum of the invocations is the configured total.

After determining how the application interacts with the ADT, the application generator finally creates a set of applications with interchangeable data structures, based on the replacement limitations described in Table 4. This is achieved by simply specifying an actual data structure in the ADT, which is a C++ template. Thus, the behavior of the synthetic applications is exactly same, i.e., the only difference

is that they have a different data structure.

Since the random numbers completely determine every behavior of a data structure, a different sequence of random numbers leads to different interactions with the ADT, and thus different sets of applications. With that in mind, the application generator must use a randomization method that has a sufficiently low probability of generating equivalent random sequences.

4.4.3 Training Framework

input : $data\_structures$ from config
input : $need\_more\_sets$ from config
output: $seed\_ds\_pairs$ - pairs of seeds and data structures
;
Map<seed, DS> $seed\_ds\_pairs \leftarrow \emptyset$;
Map<DS, time> $runtime \leftarrow \emptyset$;
while $need\_more\_sets$ **do**
    $seed \leftarrow Time()$;
    forall $DS \in data\_structures$ **do**
        $A \leftarrow Compiler(AppGen(seed, DS))$;
        $A()$ // run;
        $runtime[DS] \leftarrow GetRuntime()$;
    end
    $seed\_ds\_pairs \leftarrow seed\_ds\_pairs \cup (seed, FastestDS(runtime))$;
    $runtime[DS] \leftarrow \emptyset$;
    update $need\_more\_sets$;
end

Algorithm 1: Training Framework Phase-I

Figure 18 and Figure 19 show how the training framework of Brainy functions based on the application generator. The training consists of two phases, each of which are iterative processes. As detailed in Algorithm 1, the first phase (Phase-I) consists of iterations of generating sets of synthetic applications with the same behavior but different data structures using the application generator. The applications are compiled, run on the target machine, and the execution time is measured to determine which data structure is the best for each application. Then in

$seed\_ds\_pairs$, the best data structure is recorded together with the seed value used to generate the set of the applications [2].

Updating $need\_more\_sets$ is complex as there is no intervention or effort to generate applications that are best for a specific data structure; so after many iterations some data structures will have more "best" applications than others. Brainy stops Phase-I when a certain number of applications, e.g., ten thousand, is best for each data structure and switches to the next step (Phase-II). This threshold number is adjustable, and it is possible to use a different threshold for each data structure through the configuration file. It is important to note that the Phase-I is very fast since it does not perform any expensive profiling to extract features. Thus, measuring the applications' execution time to determine the best data structure has minimal overhead.

input : $data\_structures$ from config
input : $seed\_ds\_pairs$ from Phase-I
output: $train\_set$ - training data for model
;
Map<DS, Map<features,DS>> $train\_set \leftarrow \emptyset$;
forall $seed \in seed\_ds\_pairs$ **do**
  forall $DS \in data\_structures$ **do**
    $A \leftarrow Compiler(AppGen(seed, DS), Instrumentation)$;
    $A()$ // run;
    $features \leftarrow GetFeatures()$;
    $train\_set[DS] \leftarrow train\_set[DS] \cup (features, seed\_ds\_pairs[seed])$;
  end
end

Algorithm 2: Training Framework Phase-II

In Phase-II, the application generator replays the executions of the applications in Phase-I by taking the seed value recorded in Phase-I (as using the same seed guarantees producing the same sequence of random numbers in most pseudo-random number generators). Note, using seeds is but one way of retaining the

[2]Brainy records the best data structure only if it is 5% or more faster than any another. This prevents a data structure, which is barely the best, from being selected as an alternative.

applications between phases. That is, the applications are regenerated, and therefore Brainy can execute millions of training applications without an explosion in disk space. As shown in Algorithm 2, this phase iterates through the recorded seed values ($seed\_ds\_pairs$), regenerates the applications, and compiles them with additional instrumentation, specifically a modified STL library that has profiling for data structures. With this profiling, all of the software and hardware features can be collected during program execution. The profiling data structures record the features in a designated training set file according to the type of the data structure. $train\_set$ is updated with the collected features and the best data structure as observed in Phase-I. Again, the applications generated in each iteration have the exact same behavior, and the only difference between them is the data structure implementation. This iterative process stops when all the seeds are consumed. At the end, each data structure's training set file is fed into the machine learning tool to train the corresponding model.

In addition, Brainy's training framework is flexible. When long training time is unacceptable, users can specify that training occur for only a small number of training applications for each data structure, e.g., train only 1000 applications for each data structure. The two-phase training framework can prevent extra applications generated in Phase-I from being fed into Phase-II which performs a time-consuming feature profiling. E.g., if Phase-I generates 1500 and 1000 applications for `vector` and `list`, respectively, Phase-II does not accept the rest 500 `vector` applications. In this way, the framework can dramatically reduce the training time.

One might suggest simply using real applications to train the machine learning algorithm. However, this approach is neither practical nor plausible. Assume that there is a good real application which clearly shows `list` is better than `vector`. Nevertheless, this real application just shows one particular case among millions

of situations where `list` outperforms `vector`. For effective data structure selection, the training process must cover as many cases as possible, so that the machine learning model will yield accurate prediction results for unseen applications, which are practically infinite. That is, if the model just learns a few cases where one data structure is better than another, the resulting data structure selection is very likely to be inaccurate for real applications that were unseen in the training process.

The application generator is a reasonable approach for modeling the myriad cases required for accurate machine learning predictions. Furthermore, this framework for modeling has further advantages over real applications (or hand constructed benchmarks) by not being tied to current implementations / architectures. Otherwise, every variation to any part of the system would potentially require constructing a new set of applications. Therefore, it is desirable that the framework can automatically produce training examples tuned to the specific architecture within a reasonable time.

4.5 Artificial Neural Network (ANN)

Several machine learning techniques have been proposed over the last few decades, and it remains a question of great debate as to which machine learning technique is optimal for a given classification problem. The accuracy of the machine learning technique is inherently dependent on the characteristics of the data set. For example, Artificial Neural Network and Support Vector Machine generally perform better when the features are continuous and multicollinearity is present. They can both deal with a case where relationship between input and output features is non-linear[3], i.e., data are not linearly separable. [98, 73].

[3]Support Vector Machines can also address this case with the help of transformed feature space. A linear separation in the transformed feature space corresponds to a non-linear separation in the original space [73].

The features generated by instrumentation code show both linear and non-linear characteristics. Brainy exploits Artificial Neural Network (ANN), since it is robust to noise as well as effective for linear and non-linear statistical data modeling [55]. This seems an appropriate approach in that data structure selection is a highly complex problem domain and its training examples may have considerable noise and model-bias, thereby hurting the prediction accuracy. The training of the ANN model in this work leverages a back-propagation algorithm [114].

The ANN model predicts the alternative data structure that achieves the best performance in replacing the original data structure in an application. The target data structures, determined in Section 4.3, have their own ANN model as shown in Figure 17. That is because the list of features necessary for predicting the best data structure type is different between data structures. For example, `vector` suffers from `resizing` when its capacity is full, but `list` does not. In particular, there is another model for `vector` and `list` to address the situation when they are used in an order-oblivious manner (where insertion order has nothing to do with data organization in the data structure). When they are used in this manner, `vector` and `list` can be replaced with `hash_set` or `set`. When the underlying hardware system is changed, the ANN models for data structures should be trained and learned again for the new microarchitecture, possibly with a new set of training examples. This is achieved with the help of the application generator.

4.5.1 Feature Selection

It is important to determine which subset of features to collect for the training examples. By selecting only the most relevant features, the machine learning model will be more accurate and the learning process will converge faster. Initially, most of interface functions of a data structure and, if available, how much work is done on their invocation are collected through instrumentation code. This work calls the

latter a cost of each interface invocation. For example, `find` has a cost to model how many data elements are accessed until the search operation is finished. Similarly, for `erase` and `insert`, their costs represent how many data elements, located after the insertion and removal point, are moved backwards or forwards. Along with these software features, hardware features are also considered to make the model aware of underlying hardware architecture.

Initially, we collected the numbers on L1 and L2 caches, TLB, retired instruction, page faults and processor clock cycles, and so on. Especially, this work omits some features such as L2 cache misses, TLB misses, OS page faults, and bus utilization, since manual feature selection empirically shows that these features rarely affect the prediction of the best data structure. Since all the code to be executed becomes entirely different after data structure replacements, Brainy uses hardware features just to capture how the original data structures show certain behaviors useful for data structure selection.

To perform the feature selection, this work leverages the evolutionary approach based on genetic algorithm due to its success especially for large dimensions of features [121]. This approach represents a given subset of features as a *chromosome*, a binary string with the length of the total number of features. In the *chromosome*, each binary value represents the presence of a corresponding feature. The population of chromosomes (different feature selection candidates), evolves toward better solutions. Meanwhile, *mutation* in the genetic algorithm prevents the evolution from getting stuck in local optima, helping to approach the global optimum. In particular, this work constitutes the *chromosome* as real-valued weights, instead of binary value, that show which feature has more impact on the resulting model instead of binary values [62, 59].

Table 6 shows the top five features with the highest weight for each ANN model. For each data structure, the order of features shown in the table follows

Table 6: Selected features for each data structure

vector	order-oblivious vector	list	order-oblivious list	set	map
resizing	br_miss	iterate	find_cost	find_cost	L1_miss
insert	find_cost	push_front	find	L1_miss	data-size / cache block-size
br_miss	L1_miss	L1_miss	L1_miss	data-size / cache block-size	br_miss
insert_cost	resizing	insert	erase	find	insert_cost
iterate	erase_cost	erase_cost	data-size / cache block-size	insert_cost	find_cost

the decreasing order of the weights, e.g., the first low corresponds to the features with the highest weight.

The most important features to decide whether `vector` should be replaced, no matter if it is order-aware or order-oblivious, contain the number of `resize`s, that is performed on data insertion when the size of `vector` is full. It is interesting that a misprediction rate of conditional branches belongs to the important features. This results from the fact that such a branch misprediction can model exceptional behaviors of data structures, e.g. invoking `resize` on `insert` operations of `vector` and `hash table`. In other words, data insertion to the data structures does not suffer from performing `resize` for most of time if the capacity of the dynamic array is not full. Note that once `resize` is invoked due to insufficient capacity, it takes a while to see the recurrence of another `resize`. The reason is that `resize` extends the capacity, in case there are many more data insertions to again fill the array. In the `insert` function, a conditional branch instruction determines whether `resize` is invoked. The branch predictor could fail to correctly predict the branch instruction for this uncommon path, which is a taken branch to call `resize`. This is justified in Figure 20 where the X-axis corresponds to the branch misprediction rate while the Y-axis to the `resize` ratio (%) among the total interface invocations.

It turns out that `insert` and `insert_cost` are relevant features for `vectors`. This makes sense since these features capture how much `vector` suffers from shifting data after the insertion point. The same goes for why `erase_cost` is relevant for the order-oblivious `vector`. In particular, when `vector` is used in the

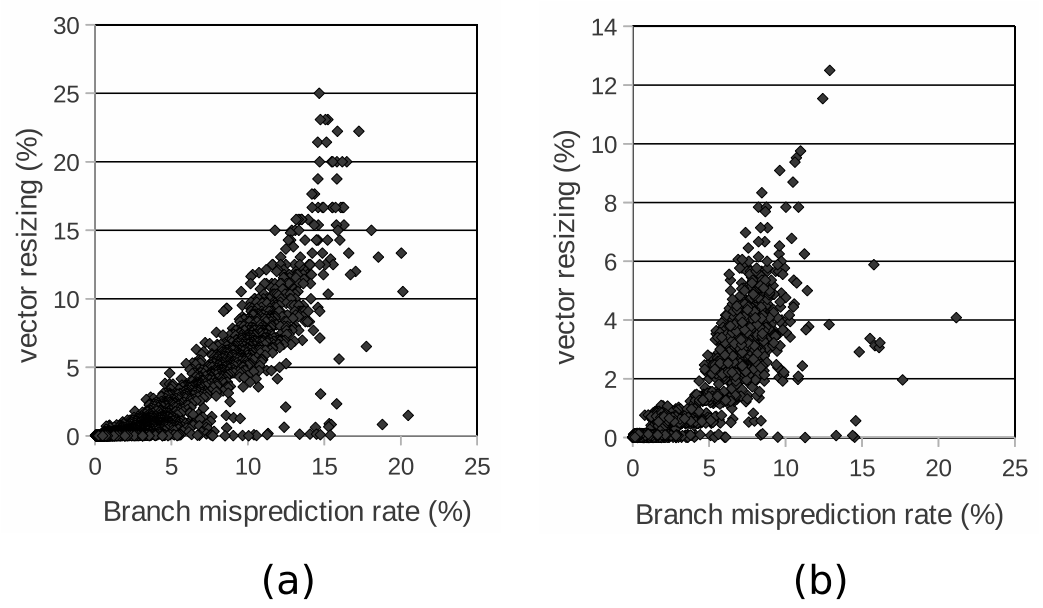


Figure 20: Correlation between conditional branch misprediction and vector `resizing` when the data structure is order-aware (a) and order-oblivious (b)

order-oblivious manner, `find` is a relevant feature. Note that in this case, there is no explicit iteration operation, thus every data access is performed by `find`.

For order-aware and order-oblivious `lists`, L1 cache miss rate is a relevant feature. It can be thought that the miss rate would capture how the nodes of the linked list fit into a cache block. Again, for the order-oblivious `list`, `find`-related features are relevant. In particular, `push_front` is relevant when `list` is used in the order-aware manner. This is understandable given how frequently data insertion occurs at the beginning of data structures, which can guide whether `vector` or `deque` is an appropriate alternative.

For `set` and `map`, `find`-related features are most relevant, as their data structure selection highly depends on how frequently `find` is performed and how many data elements a `find` operation accesses. Again, the `insert_cost` and `find_cost` represent the number of data elements accessed while the corresponding operations reach the insertion point and the search location, respectively. In

addition, L1 cache miss rates and data element size per cache block size can capture how long the latency of each data element is on the `find` operation. Thus, they can quantify the cost of data accesses involved in `find` operations.

4.5.2 Limitation

While Brainy captures many useful properties with synthetic applications created by the application generator, it also has a limitation that leaves room for future improvement. The synthetic applications might not accurately model the impact of other parts of a real application on the microarchitectural state, e.g., the L1 is polluted by data in intervening instructions. However, it should be noted that Brainy is aware of such a microarchitectural behavior, and possibly another synthetic application can capture the polluted L1 cache behavior.

Even with these drawbacks, it turns out that the training with the synthetic applications can "cover" real applications. That is it is conjectured that the behaviors exhibited in actual execution would be a subset of training behaviors therefore hoping that the actual execution model would be subset of the constructed one. Section 4.6 demonstrates that for real-world applications, Brainy can consistently select optimal data structures across input and architectural changes.

4.6 Evaluation

In order to evaluate the effectiveness of Brainy, we implemented it as a part of C++ Standard Template Library (STL) for GCC 4.5 [131]. To access hardware performance counters, we used PAPI [44]. Especially, to show Brainy's accuracy across different inputs, we selected a set of C++ applications where the best data structure varies on input changes. The data structure selection experiments were performed on two different systems that have Intel Core2 and Intel Atom microarchitectures, respectively. The detailed system configurations are described in Figure 21.

In the next sections, we first validate Brainy's data structure selection models.

Desktop	
CPU	Intel Core2 Quad Q6600 2.4 GHz
Caches	4 X 32 KB L1 data, 2 X 4 MB L2 unified
Memory / DISK	2 GB SDRAM, 200 GB HDD
Operating System	64-bit Ubuntu Desktop 8.04
Compiler	GCC 4.5 with libstdc++ 4.5.0
Laptop	
CPU	Intel Atom N270 1.6 GHz with HyperThreading
Caches	32 KB L1 data, 512 KB L2 unified
Memory / DISK	512 MB SDRAM, 8 GB solid state disk (SSD)
Operating System	32-bit Ubuntu Netbook Remix 9.10
Compiler	GCC 4.5 with libstdc++ 4.5.0

Figure 21: Target systems configurations

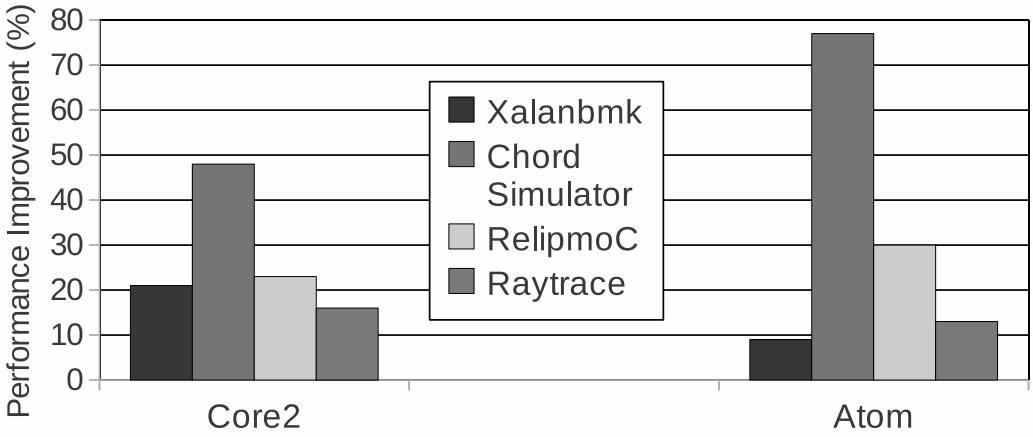


Figure 22: Performance improvement Brainy achieved

Then, we show four case studies with real-world applications. In the first two applications, the optimal data structures vary across inputs and even microarchitectures (Section 4.6.3). Thus, they show the difficulty of accurate data structure selection. In the next two applications, the optimal data structures are rarely affected by input and microarchitecture changes. Thus, we show their results briefly compared to the first two applications.

Figure 22 summarizes the performance improvement of each application obtained from Brainy's data structure replacement. In cases where the optimal data structure varies across inputs, only the best performance result Brainy achieved

appears in the figure. Brainy achieved an average performance improvement of 27% and 33% on Core2 and Atom microarchitectures, and up to 77% for some case (Section 4.6.3).

4.6.1 Model Validation with an Application Generator

Validating Brainy's data structure selection models leverages the application generator. For an accurate and fair evaluation, the application generator newly produces 1000 random applications for each data structure model. Note that all these random applications have never been seen by the models, i.e., the model validation is performed with completely new applications. Thus, the applications here are not the ones used to train the models. The accuracy is calculated as follows;

$$accuracy(\%) = 1 - \frac{The\ number\ of\ mispredictions}{1000} \tag{11}$$

Figure 23 shows how accurate the prediction results of each data structure model are for the 1000 applications on the Core2 and Atom microarchitectures. Overall, for Core2 microarchitecture, the accuracies of models are between 80% and 90%. This is impressive in that the 1000 applications for each model capture a variety of behaviors of data structure usages, thus the best data structure is quite different across the applications. It needs to be noted that each data structure model attempts to select the best data structure among many replaceable data structures as described in Table 4. For instance, the model for `vector` selects the best data structure among possible six candidates, when it is used in the order-oblivious manner. For Atom microarchitecture, the accuracies of models are between 70% and 80%. This is enough to effectively predict the best data structure of a real application as described in the next section.

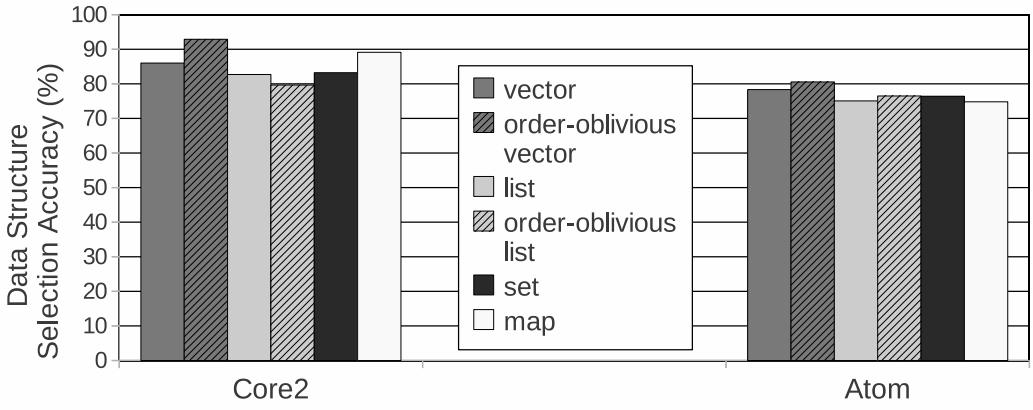


Figure 23: Accuracy of data structure selection models; for the same data structure, there are two different models for Core2 and Atom microarchitectures, respectively.

4.6.2 Xalancbmk

Xalancbmk is an open source XSLT processor that performs XML to HTML transformations. It takes as inputs an XML document and an XSLT style sheet with detailed instructions for the transformation. The program maintains a string cache comprised of two levels, *m_busyList* and *m_availableList*, `vector`s. When a string is freed in *XalanDOMStringCache::release*, it moves the string to the *m_availableList*, provided it is found in the *m_busyList*. To determine whether the string is found in the latter list, the data structure, `vector`, performs `find` operations. In general, these operations are often recurring, but the frequency of performing them is varying across program inputs. In addition, each input brings about different search patterns.

To define the accuracy of Brainy for data structure selection, the evaluation process leverages comparison with the Oracle scheme which is empirically determined across program inputs on each microarchitecture. If the resulting data structure selection agrees with the Oracle's, the result are considered accurate.

In addition, the evaluation compares Brainy with Perflint, the state-of-the-art

data structure advisor that relies on hand-constructed models [87]. On each interface invocation, Perflint assigns the cost taking into account traditional asymptotic analysis. As an example, for the cost of a `find` operation among N data elements, `vector` leverages average case for linear search, i.e., '$3/4N$', while `set` uses '$\log N$' for binary search[4]. Each cost is multiplied with a coefficient value, which is determined by linear regression analysis for execution time, and accumulated whenever the interface function is called. In particular, Perflint provides the hand-constructed model for `vector-to-set` replacement while `vector-to-hash_set` is not supported. Each interface invocation of the original data structure (`vector`) updates the costs of both `vector` and `set`. Based on comparing the accumulated costs at the end of program execution, Perflint selectively reports the alternative data structure.

Figure 24 shows execution times of three selected data structures, `vector`, `set`, and `hash_set` with those schemes. The ideal data structure selection (Oracle), i.e., `vector` is the best for a train input while `hash_set` for test and reference inputs, are identical on both microarchitectures. Especially, `set` performs differently on the two microarchitectures. That is, for test and reference inputs, `set` outperforms `vector` on Core2 microarchitecture while the data structure replacement to `set` does not achieve significant performance improvement on Atom microarchitecture.

Figure 25 shows the results of each data structure selection scheme for the two different microarchitectures. Baseline represents the original data structure in the figure. According to the Oracle, for test and reference inputs, the original data structure, which is `vector`, is desired to be replaced with `hash_set` for better performance. The reason is that the data structure executes many search operations. However, for a train input where `hash_set` is suboptimal, `vector` is the

[4]For binary search, the average and worst cases are exactly the same.

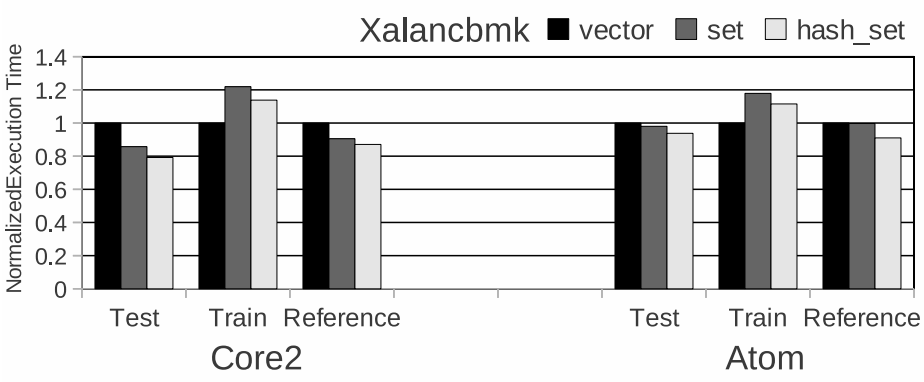


Figure 24: Normalized execution time across different data structures; The baseline execution times (in second) are on Core are 3s, 74s, and 234s for test, train, and reference, respectively. On Atom, the baseline execution times for these inputs are 18s, 611s, and 1345s, respectively. Brainy selects the best data structure for each input of Xalancbmk

Input Size	Selection Schemes	Reported Best DS	
		Core2	Atom
Test	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`hash_set`	`hash_set`
	Oracle	`hash_set`	`hash_set`
Train	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`vector`	`vector`
	Oracle	`vector`	`vector`
Reference	Baseline	`vector`	`vector`
	Perflint	`set`	`set`
	Brainy	`hash_set`	`hash_set`
	Oracle	`hash_set`	`hash_set`

Figure 25: Xalancbmk's data selection results on Core2 and Atom microarchitectures.

best data structure. This is not easily understandable and rather surprising. According to the profiled features with instrumentation code of Brainy, the application invokes the `find` function more than 60 millions times for a train input as well as for a reference input. On top of that, the train input causes the application to `erase` the first data element from the head of the dynamic array almost 30 times more frequently than the reference input does, which is pretty problematic for `vector`. On the other hand, for the test input, the application achieves the best performance with `hash_set` in spite of a relatively small number of `find` function

invocations, which is about thirty-seven thousand. Thus, accurate data structure selection is very difficult for this application.

With the help of the profiled feature results, it turns out that `find` operation is much more dominant compared to the problematic `erase` operation. What happened behind the scenes related to the `find` operation is that the number of data elements the operation touched is varying across program inputs. This is mainly due to the change of search patterns across inputs. Table 7 presents more detailed information about this situation. This implies that building an accurate hand-constructed model would be much more difficult.

Table 7: The number of `find` invocations and the total number of touched data elements for all the invocations across program inputs.

Input Size	`find` invocations	Touched data elements
Test	37,594	32,804,644
Train	62,438,422	2,569,120,180
Reference	67,720,063	89,454,229,684

For the training input, a majority of `find` operations succeed in searching the designated data element in the very beginning of the dynamic array of the original data structure, `vector`. In this case, `hash_set` just causes extra memory consumption compared to `vector`. It is desirable to force the application not to pay for complex operations such as maintaining hash buckets which is not really necessary, thus `vector` is preferable to `hash_set`. Brainy can recognize the search pattern based on `find`-related features as described in Section 4.5.1. Together with considering other software and hardware features profiled, Brainy correctly reported the same results as the Oracle across different inputs for the both microarchitectures.

Meanwhile, Perflint failed to consistently report accurate prediction results for the best data structure, even if it only needs to perform a binary decision between `vector` and `set`. For the train input, Perflint incorrectly reported that `set` is

preferable to `vector`. This is problematic because the resulting data structure replacement to `set` causes performance degradation on both microarchitectures as shown in Figure 24. For the reference input, Perflint reported that `set` is preferable, which only works on Core2 microarchitecture, i.e., replacing `vector` with `set` achieves little performance improvement on Atom microarchitecture. Again, Brainy selected the optimal data structures consistently across all the program inputs on both microarchitectures.

4.6.3 Chord Simulator

This application is an open source simulator for Chord, a distributed lookup protocol to locate Internet resources. The main work of the simulation is to send query requests for a certain resource over the network and to record if the lookup fails by checking the response to the query. Whenever the response is received, the simulator drops the message, which corresponds to the resource of the response, in a pending list of routing messages. The search performance thus translates to the simulation time reduction. In particular, determining the message to be dropped performs *std::find_if* on the pending list, which is implemented using `vector`, checking an ID field of each message structure. Thus, the vector can be replaced with map-like data structures using the ID field as its key.

Brainy suggested to replace the original `vector` with `map` or `hash_map`, according to different inputs. In the application, the optimal data structure varies across different inputs on both microarchitectures, as shown Figure 27. It is important to note that for the Large input, the optimal data structures on both microarchitectures do not agree with each other, i.e., `vector` is optimal on Core2 whereas `map` performs the best on Atom. This shows the difficulties of the data structure selection in the application. Overall, Brainy correctly reported the same results as the Oracle across different inputs and microarchitectures. It needs to be noted that

when `vector`, the original data structure, is optimal, Brainy correctly selected this data structure. Figure 26 shows the performance results of different data structures across different inputs and microarchitectures. The configuration of the graph and the table follows the one in the previous section.

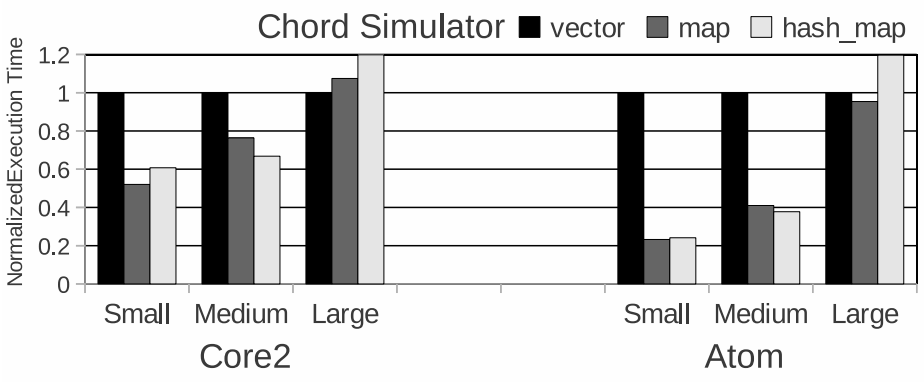


Figure 26: Normalized execution times across different data structures: the baseline execution times (in second) on Core2 are 9s, 19s, and 306s for test, train, and reference, respectively. On Atom, the baseline execution times for these inputs are 47s, 203s, and 2952s, respectively. Brainy selects the best data structure for each input of Chord Simulator.

Input Size	Selection Schemes	Reported Best DS	
		Core2	Atom
Small	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`map`	`map`
	Oracle	`map`	`map`
Medium	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`hash_map`	`hash_map`
	Oracle	`hash_map`	`hash_map`
Large	Baseline	`vector`	`vector`
	Perflint	`map`	`map`
	Brainy	`vector`	`map`
	Oracle	`vector`	`map`

Figure 27: Chord simulator's data selection results on Core2 and Atom microarchitectures.

Again, we compared Brainy with Perflint[5]. Perflint selected `map` for all combinations of inputs and mircroarchitectures. However, for the Large input on Core2,

[5]Since Perflint does not support `vector-to-map` replacement explicitly, this work considers its suggestion of `set` as the replacement to `map`. We believe that the implementation of the replacement should exactly follow the manner that `vector-to-set` is implemented.

`map` performs worse than the original data structure, `vector`. Perflint's suggestion causes performance degradation in this case. In contrast, Brainy consistently selected the optimal data structures for all combinations of inputs and microarchitectures.

4.6.4 RelipmoC

RelipmoC is an open source translator that converts i386 assembly code to C code, i.e., a decompiler for i386 assembly. It analyzes the input assembly code and builds a list of basic blocks implemented using STL `set`, thus a red-black tree. On the `set` data structure, it performs data flow and control flow analyses to extract high level expressions, and to recover program constructs, e.g., loops and conditional statements, along with the information about their nesting level. It frequently checks if a basic block belongs to the program constructs which are normally a list of basic blocks. In the meantime, `find` and iteration operations are executed many times for short lists and long lists of basic blocks, respectively. Brainy suggested replacing `set` with `avl_set`, the implementation of which is an AVL tree. By conducting the suggested replacement, we improved the execution time of the application on Core2 and Atom microarchitectures by 23% and 30% on both microarchitectures, respectively. The baseline execution times (in seconds) of this application on Core2 and Atom are 41s and 120s. We could not compare Brainy with Perflint since it does not support any replacement for `set`.

4.6.5 Raytrace

This application draws a 3D image of groups of spheres using a ray tracing algorithm implemented in C++ STL. The spheres are divided into groups that use `list` to store them. The main computation of the program occurs in a loop on *intersect* of each group object. First, the intersection calculation is performed for each group of spheres. If a ray hits the group, it is subsequently performed for its

spheres (scenes). Thus the `list` is heavily accessed and iterated during the ray tracing, i.e., `vector` is much preferable. Brainy correctly suggested to replace the `list` with `vector`. By taking Brainy's suggestion, we replaced the original data structure with `vector` thereby reducing the execution time of the application on Core2 and Atom microarchitectures by 16% and 13%, respectively. The baseline execution times (in seconds) of this application on Core2 and Atom are 79s and 347s. This time Perflint selected the optimal data structure just as Brainy did.

4.7 Summary

Data structure selection is one of the most critical aspects in determining program efficiency. This paper presents Brainy, a novel and repeatable methodology for generating machine-learning based models to predict what the best data structure implementation is given a program, a set of inputs, and a target architecture. The work introduces a random program generator that is used to train the machine learning models, and demonstrates that these models are more accurate and more effective than previously proposed hand-constructed models based on traditional asymptotic analysis for real-world applications. The experimental results show that Brainy achieved an average performance improvement of 27% and 33% on two real machines with different processors.

CHAPTER V

MEMORY LEAK DETECTION FOR DATA STRUCTURES

5.1 Introduction

Memory management bugs are a common source of persistent errors in real-world code. Memory leaks are particularly notorious, since their symptoms and causes are insidious and hard-to-track [104, 56]. Most of the data structures are dynamically allocated in the heap area of the memory and one of the most common problems encountered for the dynamically allocated objects is the memory leaks. They occur when allocated objects are not freed, even if they are never accessed again. Since they remain allocated consuming the heap memory, they gradually affect the quality-of-service (QoS) of the system. Even worse, piled leaks eventually crash applications by exhausting system resources. Memory leaks can also result in software security/reliability problems (CWE-401) [35]. For example, many CVE entries including CVE-2013-0152/0217/1129 have detailed the problems [34], and malicious exploits have been designed based on memory leaks to launch denial-of-service (DoS) attacks [138].

In the manycore era, leaks are more common than ever in multithreaded software. When heap-allocated objects *escape* their thread, it is hard to determine when and which thread is to deallocate them. Due to the difficulties of reasoning about the *liveness* of the shared objects, programmers often end up leaving the objects allocated in the memory thereby producing leaks. Despite undergoing extensive in-house testing, leaks often exist in deployed software and show up in customer usage [20, 17]. In fact, they are common causes of bug reports for production software [101, 2].

With the advent of cloud services that allow customers to deploy various services in the datacenter, memory leak detection is one of the most critical issues in datacenters. Several reasons drive this movement. First, the threat of service downtime due to leaks has been a constant concern in datacenters [123, 8]. Such a service-level-agreement (SLA) violation leads to the penalty, e.g., a reduction in fees [47].

Second, since each machine in the datacenter supports multiple services in general, one leaking application can threaten the QoS and the reliability of every service running on the same machine. I.e., leaks impact not only the leaking application but also all the others, due to the limited amount of available system memory [31].

Third, memory leaks directly affect the datacenter operational cost; the fact that the service applications can be leaky puts significant pressure on resource overprovision in the datacenter. Once memory is actually leaking, the datacenter ends up consuming more and more resources, e.g., co-locating fewer services in a machine in the datacenter, to deliver as promised in the SLA.

Lastly, the datacenter provider needs not only to detect the threat of leaks but also to correctly attribute it to the leaking application; just consuming large memory should not be blamed unless the application is leaking. That is necessary to adjust the SLA and better support it rather than to blame for the memory leak. E.g., after fixing the leak, the customer can run the service with a lower cost while the provider can allocate less resource to it. Thus, effective memory leak detection can improve the datacenter ecosystem by helping the provider as well as the customers.

Unfortunately, existing tools [56, 104, 93, 31, 28, 106, 20] cannot be used in datacenters for many reasons. First, the tools cannot meet the QoS demand due to their high overhead. While state-of-the-art tools leverage sampling techniques to track

accesses to heap objects [28, 106], the resulting overhead is still unacceptable, e.g., 9.72x slowdown and more than 70% dynamic memory increase for heap-bound applications.

Note that such memory-consuming approaches including [106, 109] are prohibitive in datacenters. In reality, even 5% increase of heap size due to faster memory allocation makes it impossible to use the memory allocator in enterprise systems. Apart from that, *it just makes no sense to spend more memory for less leak*.

More importantly, existing tools are neither systematic nor automated. Their leak determination relies on a manually-set threshold. That is, user intervention is required for each service, and even worse such a high cost will have to be paid anew on environmental change, e.g., SLA adjustment or microarchitecture change. It is unrealistic for datacenter providers to ask the customer to provide the threshold for every service/SLA/architecture combination.

The lack of a methodology to determine the threshold forces users to do that properly, or ends up blindly applying a fixed threshold to those applications that have different characteristics. As a result, existing tools can falsely blame non-leaking objects or miss real leaks. I.e., the tools inherently vulnerable to false positives and negatives.

Given all this, there is a compelling need for a practical memory leak detection tool usable in datacenters. With that in mind, this paper presents the design and implementation of Sniper to effectively detect memory leaks in C/C++ production applications. It leverages instruction sampling using performance monitoring units (PMU) in processors to track accesses to heap objects without significant overhead. It also offloads most of time- and space-consuming work, e.g., tracking heap organization and searching for the heap object accessed by a sampled instruction. To achieve this, Sniper uses a trace-driven approach based on the combination of a lightweight heap trace generation and an offline trace simulation.

During program execution, Sniper records full traces of malloc/free as well as sampled PMU traces. The offline simulator then analyzes those traces and calculates the *staleness* of heap objects, which is a clue to potential memory leaks. I.e., the simulator replays the program's heap-related activities, thereby catching every leak occurred during the program execution. In this way, Sniper rarely increases the execution time and the memory space at runtime. The takeaway is that the same mechanism is applicable to multithreaded program with no synchronization overhead.

In particular, Sniper's leak identification is very accurate. Rather than relying on *ad hoc* efforts which require user intervention, Sniper provides a systematic and accurate methodology to identify leaks. The key idea is to view memory leaks as anomalies. Sniper's anomaly detection is not only fully automated but also application-tailored. Such a statistical analysis makes Sniper robust against false positives/negatives across applications. E.g., chances are much low that innocent objects with over-estimated *staleness* due to the sampling are falsely blamed.

Finally, Sniper neither requires recompilation nor perturbs the application execution with instruction instrumentation or memory allocator modification. Along with the low overheads, that makes Sniper work transparently to the application. In case QoS requirement becomes more stringent, it is possible to dynamically turn off Sniper taking away all the overheads.

Overall, Sniper is usable in datacenters, and therefore can observe real execution characteristics that actually cause memory leaks. The following are the contributions of this work:

- The first systematic and automated methodology for accurate leak identification based on an anomaly detection. The issue is to automatically determine the *staleness* threshold, which is an '**open problem**' in that no prior work has addressed the issue despite its importance. Our leak identification, which

is tailored not just for each application but for each allocation site as well, effectively deals with the problem.

- A new trace-driven methodology to track how an application interacts with the heap; it not only offloads heavy work to the offline simulator, but also enables the statistical analysis of leaks.

- To the best of our knowledge, Sniper is the first to effectively detect leaks for multithreaded software.

- No heap size increase and very little time overhead.

5.2 Background and Motivation

This section introduces basic concepts and terminologies used in state-of-the-art leak detection tools, and shows their limitations as well as the requirements for production use in datacenters, which drive Sniper's design.

5.2.1 Target Memory Leaks

Memory leaks are of two kinds: (1) *unreachable* memory, i.e., program cannot access it, and (2) *dead* memory, i.e., it is reachable, but the program will never use it again, thus it is not *live*. The *unreachable* leaks can be effectively addressed by garbage collection [15] and static analyses [57, 144, 26, 70]. However, the *dead* leaks are much more tricky, since it is in general undecidable to determine if certain memory will not be accessed in the remainder of the program execution. This difficulty leads to the advent of many dynamic analyses [109, 18, 147, 16, 17, 31, 20] including Sniper. The focus of this work is to detect the *dead* leaks even though Sniper can deal with the both types of leaks.

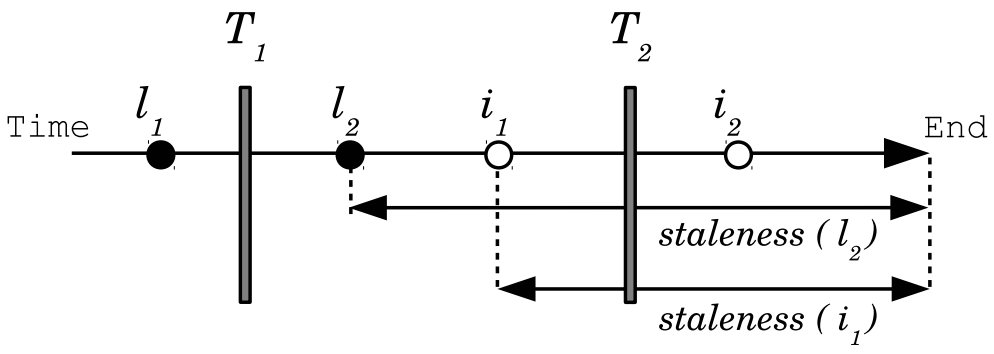


Figure 28: The determination of $threshold_{staleness}$.

5.2.2 Staleness-Based Leak Detection

Chilimbi and Hauswirth formulated the problem of memory leak detection based on 'how stale heap objects are' in their pioneering work called SWAT [28]. They define the *staleness* of an object as *how long it remains unaccessed since the last access time*. SWAT reports those heap objects whose *staleness* is greater than the length of timeout. I.e., if an object has not been accessed for a long time, it is likely to be a leak. At time t_{report}, an object o is identified as a leak if $t_{report} - t_{last\_access}(o) > threshold_{staleness}$. To track the last access with low overhead, existing tools leverage a sampling technique.

Depending on the threshold, they can end up reporting innocent objects as leaks (false positives) and miss real leaks (false negatives). Figure 28 shows the importance of accurate threshold determination. In the figure, circles on a time arrow represent the last access of heap objects whose *staleness* appears below the arrow. T_n corresponds to each threshold while l_n and i_n to leaking and innocent objects, respectively. Here, T_1 misses l_2 since its *staleness* is less than T_1, thus smaller threshold is desirable. Alternatively, T_2 correctly identifies l_2 as a leak, but falsely blames i_1 since its *staleness* becomes greater than the threshold. The ideal threshold exists between $t_{last\_access}(\mathrm{argmin}_{x \in \mathcal{L}}\ staleness(x))$ and $t_{last\_access}(\mathrm{argmax}_{y \in \mathcal{I}}\ staleness(y))$ where $_{\mathcal{L}}$ and $_{\mathcal{I}}$ are the sets of leaking and innocent objects, respectively.

However, it is practically impossible to determine the ideal threshold, since such a determination already requires perfect knowledge of what object is a leak. Note that the ideal value exists if and only if

$$\min_{x \in \mathcal{L}} \ staleness(x) \ > \ \max_{y \in \mathcal{I}} \ staleness(y) \tag{12}$$

I.e., if the inequality 12 above does not hold, then

$$\nexists \ threshold \ s.t. \ \|\{FalsePositive\} \cup \{FalseNegative\}\| \ = \ 0$$

meaning that there is no threshold to achieve perfect accuracy. In reality, since existing tools leverage a sampling technique thereby over/under-estimating the staleness, the inequality 12 is likely to fail meaning that they suffer from false positives/negatives.

Note that the ideal *staleness* threshold should be different across applications. That is, the leak determination should be application-specific. In particular, this work shows that even if the inequality 12 does not hold in the first place, Sniper can still achieve good results with the help of its context-sensitive leak detection (See Section 5.3.5.2).

5.2.3 The Impact of Staleness Threshold

In a sense, the *staleness* based leak detection is an intuitive view of memory leakage problem which attributes stale objects as the symptom of memory leaks. Thus, *staleness* thresholds play an important role for accurate memory leak detection. A high threshold, which states that an object has to be highly stale for it to be reported as a leak, will detect few leaks, but be highly precise and will not report innocent objects as leaks. On the other hand, a low threshold, which states that small *staleness* is enough for the leak identification, will report relatively many objects as leaks. Here, most leaks would be detected, but such a threshold may also falsely blame many innocent objects as leaks. Therefore, a good *staleness* threshold

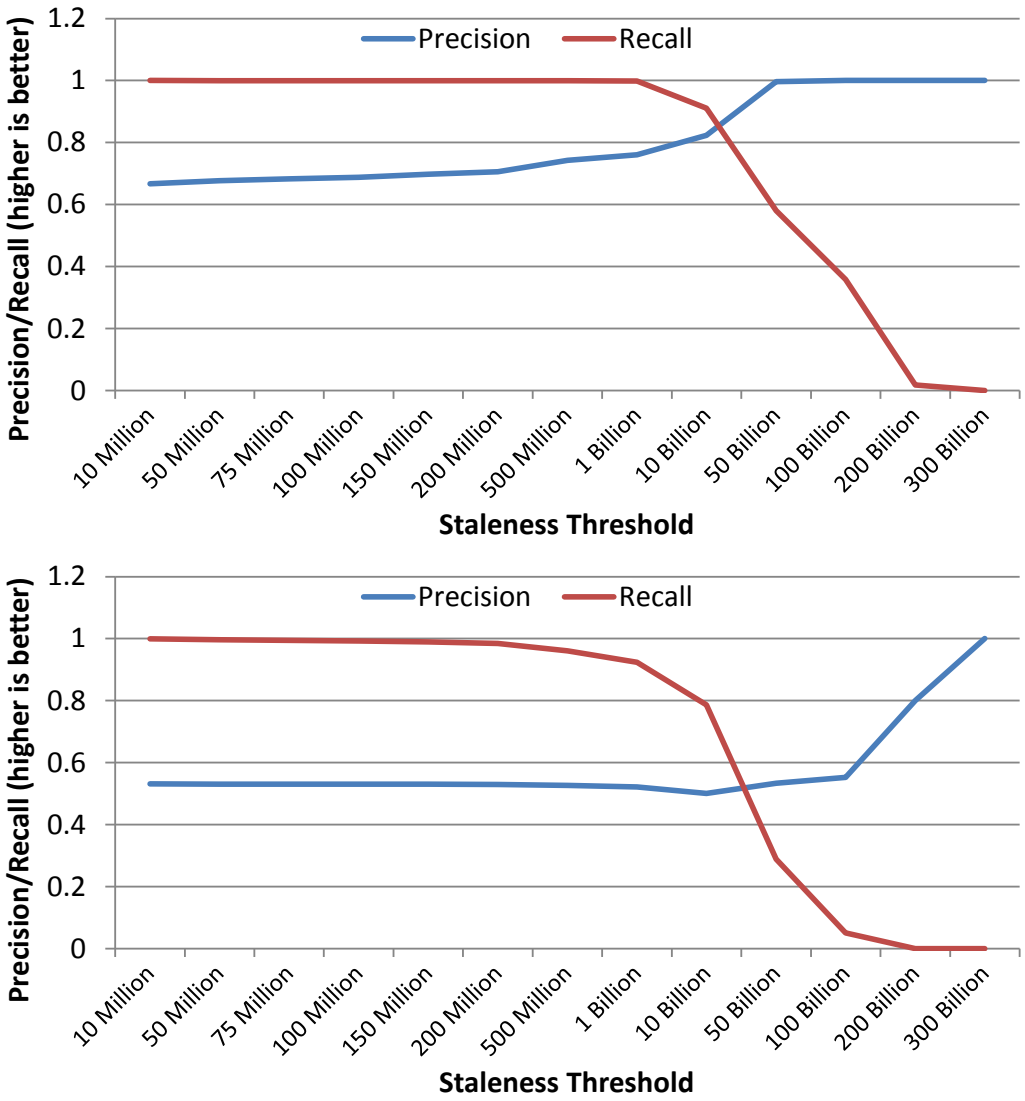


Figure 29: The accuracy tradeoff of *staleness* thresholds on *astar* (above) and *xalancbmk* (below).

should make a good balance between the two extremes so as to detect many real leaks but not to generate too many false positives which will waste the user's time inspecting the cause of falsely reported leaks.

Empirical Evaluation of SWAT's Staleness Thresholds To evaluate the efficacy of SWAT's *staleness* approach and the impact of different *staleness* thresholds, we implemented SWAT using the LLVM toolset [77] and tested it with various *staleness*

thresholds. Chilimbi and Hauswirth suggest using *idleGt1Billion* threshold [28]. However, it is doubtful whether the recommended threshold works well across applications, since the threshold came from an empirical evaluation and rather than analytical reasoning.

Our empirical results show that this concern turns out to be true impacting the accuracy of SWAT. Figure 29 depicts the precision/recall tradeoff for two SPEC2006 applications, *astar* and *xalancbmk*. The figure shows that for *astar*, there is a chance to improve the accuracy by carefully adjusting the threshold, i.e., till the *staleness* threshold reaches 1 billion, the precision increases without sacrificing the recall. On the contrary, the same does not hold for *xalancbmk*, i.e., a higher precision comes at the cost of poor recall. Thus, *astar* benefits from a relatively higher *staleness* threshold whereas *xalancbmk* from a relatively lower *staleness*.

We found three lessons here. First, selecting the best performing threshold is not an easy task. Thus, there is a compelling need to automatically determine an appropriate *staleness* threshold. Second, even if users successfully select the best *staleness* threshold for one application, the threshold tends to be suboptimal for different applications ending up with the accuracy degradation. Again, the *staleness* threshold determination should be performed in an application-specific manner.

Finally, despite the concerns of the threshold determination, the *staleness* based leak detection works well in general. For both applications, carefully selected thresholds can achieve perfect precision, e.g., those objects with very high *staleness* are actually leaking. This supports the philosophy of SWAT which states that highly stale objects are likely to be memory leaks. Again, the empirical results are not that of Sniper. However, since it is built on top of the *staleness* based leak detection, the lessons learned from this empirical evaluation of SWAT can guide the design of Sniper for accurate memory leak detection.

5.2.4 Leak Detector Requirements for Datacenters

Since memory leaks are very input- and environment-sensitive [20, 17], production use is essential to observe real execution characteristics that actually cause the leaks. There are several requirements for production use in datacenters that Sniper must meet.

First, Sniper must not cause significant overhead that jeopardizes the QoS requirements of the production service, e.g., it should not increase heap size, thus memory-consuming approaches [106, 109] are prohibitive. Second, due to the variety of the service/SLA/architecture combinations and their frequent change, Sniper must provide a systematic and automated methodology for leak determination. Third, it has to be precise; it should not blame an application for the falsely-reported leaks while real leaks must be detected. Lastly, Sniper should be able to effectively deal with multithreaded software. Otherwise, it would be considerably less useful.

Unfortunately, existing tools are not usable for production use due to their inabilities to meet those requirements. In light of this, Sniper takes into account the requirements.

5.3 Sniper Design and the Details

The first goal is to provide a lightweight memory leak detection tool usable in datacenter environment. To achieve this, we identified the key sources of runtime and space overheads in *staleness* based leak detectors. (1) the instruction instrumentation to track accesses to heap objects causes high runtime overhead. (2) most of the space overhead comes from tag (meta) data that abstracts the heap objects; for each heap object, tools need to maintain the *staleness*, the allocation site, the dynamic program point that accessed the object, and the heap organization information, e.g., the address range of the object. (3) updating the *staleness* of the heap

objects causes both space and time overheads. In particular, for every sampled load, the tools need to determine if the instruction accesses a heap object. This requires searching the tag directory[1] (which is a memory-intensive data structure) for the heap object whose range embraces the target address of the load instruction.

Sniper addresses each source of the overheads effectively. To remove the heavy-weight instrumentation completely, Sniper exploits instruction sampling using hardware performance monitoring units (PMU) available in commodity processors. To reduce the space overhead due to the tag data, Sniper buffers the full trace of malloc/free and flushes each buffer into files when it is full. Similarly, Sniper maintains another buffer to keep information about PMU samples. Thus, the additional memory consumption is bound to the size of the buffers.

Sniper also offloads time- and space-consuming work of the *staleness* update to its trace simulator. Using the malloc/free/PMU traces generated at runtime as an input, the simulator performs the expensive *staleness* update offline. That way the memory-hungry tag directory and the space needed for the *staleness* are no longer necessary at runtime. Instead, it is during the offline simulation that a tag directory is constructed and searched for the *staleness* update. In this manner, Sniper minimizes both time and space overheads during program execution by offloading much of the work.

Another goal of Sniper is to provide a systematic and automated methodology for precise leak determination. Again, the lack of the methodology ends up blindly applying a fixed threshold to all the applications which may differ significantly in their behaviors. Sniper leverages a couple of observations; (1) one-size-fit-all threshold does not exist even within the same application, i.e., multiple thresholds should be carefully determined according to application characteristics. (2)

[1]For Java program, the search is not necessary because Java allocates a header, which can store information such as *staleness*, for each requested dynamic memory.

separating objects based on program context where they are created, and then performing leak detection on the separated sets can improve the accuracy, i.e., the inequality 12 is more likely to hold. In short, the leak determination methodology should be tailored for each allocation-site as well as each application.

With that in mind, Sniper leverages a statistical analysis on the trace information as well as detailed results of the trace simulation. In particular, this work reformulates the problem of memory leak detection as that of anomaly detection. Thus Sniper views memory leaks as anomalies. The reason for this is that the *staleness* of a leaking object should be extremely higher than that of considerably many normal objects in the entire application or even in the same allocation site. The end result is that Sniper can automatically determine the *staleness* threshold in an application-specific manner.

Figure 30 shows a high-level view of Sniper. First, an application binary is fed into Sniper's launcher. It prepares a `ptrace` hook so that a `ptrace` monitor observes every PMU transaction from the core the application is running on. That way Sniper can collect the instruction samples without perturbing the application execution. Then, the launcher preloads Sniper's wrapper (.so) to hijack heap interfaces, e.g., malloc/free, and fork and executes the application. At runtime, the wrapper generates traces of the functions to track how the heap organization evolves. They are buffered and later recorded in files. PMU traces are recorded in a similar manner.

Once the application completes execution, all the traces are fed into a trace simulator. To extract program context information, it consults the binary analyzer. During the simulation, it tracks the interaction between the application and the heap organization as well as the accesses to the heap-allocated objects. I.e., the simulator replays the application's execution in terms of its heap usage, and updates the *staleness* of the allocated objects.

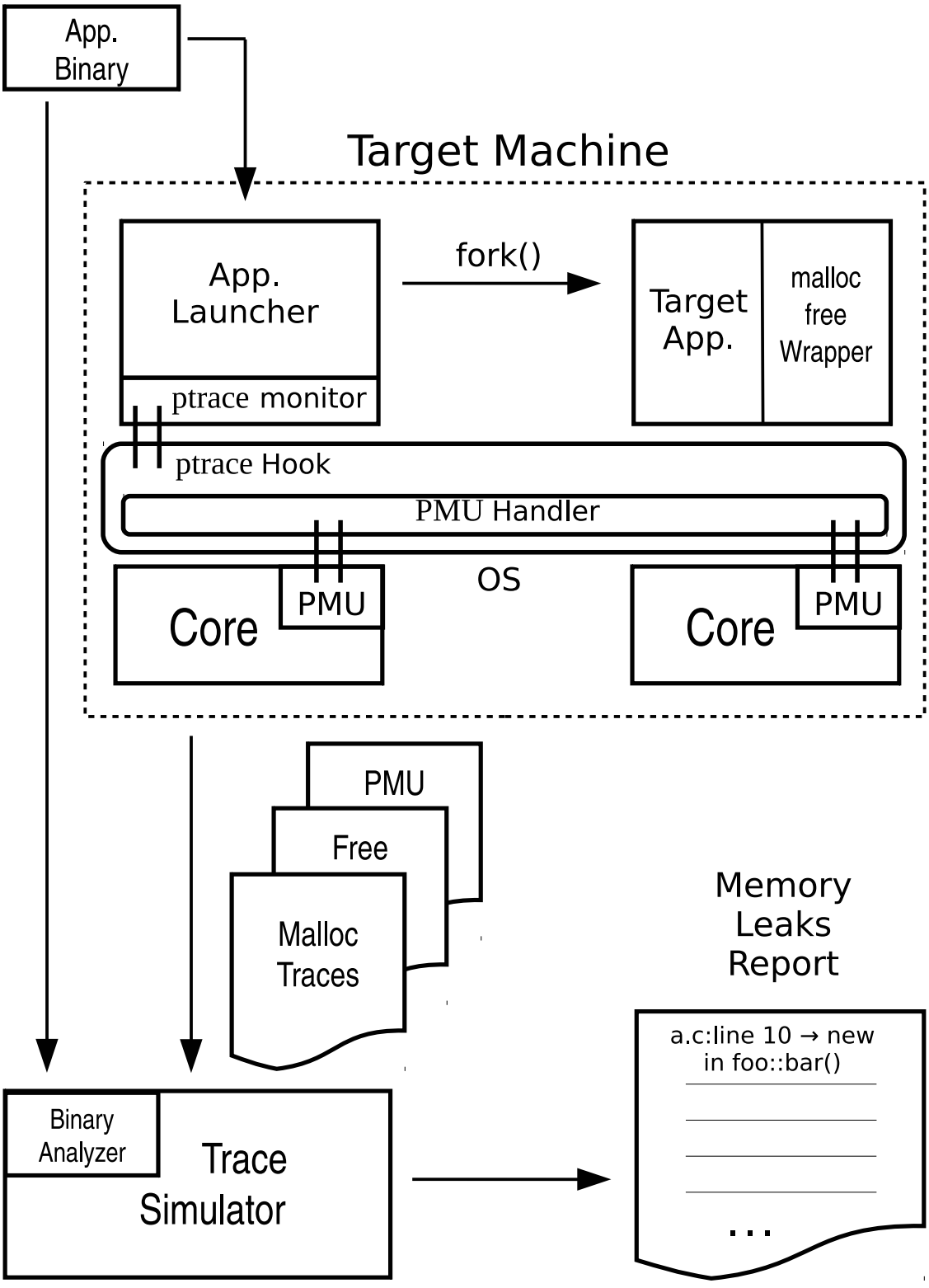


Figure 30: The Sniper Organization.

At the end of the simulation, Sniper finally reports leaks detected with its anomaly detection to Bugzilla. In particular, Sniper's report is rich with details about each object including the program context (malloc/free sites[2]) and various simulation results such as memory access/growth analysis. While prior work reports just the last access site of only leaking objects, i.e., a single instruction address, Sniper provides an snapshot of different instruction accesses to the object whether or not it is a leak thus helping developers to fix it while debugging. E.g, using the techniques found in [25], one can construct a dynamic slice to track down the offending memory allocation and find the program flow from that allocation site until the point of last use.

5.3.1 Memory Access Tracking with PMU-Based Instruction Sampling

The key to detect leaks is the *staleness* of allocated objects in that if they have not been accessed for a long time, they are likely to be leaks. By its definition, i.e., the elapsed time from the last access, the *staleness* calculation requires tracking the memory access to the objects. For efficient leak detection, it is important to collect the last access with a low overhead. With that in mind, Sniper obtains the memory access profile through the PMU without incurring a significant overhead. This section briefly presents Sniper's hardware/software internals and the related issues.

An instruction sampling is a hardware mechanism that offers a good insight into program behaviors with a very low overhead. The PMU on modern processors has a special mode called event-based sampling [60, 45]. For a given event, this mode can configure the corresponding performance counter to raise an interrupt on overflow of its value, i.e., sampling period; when an interrupt occurs, the

[2]Sniper records the return address of malloc and free at runtime. Later the offline trace simulator calculates the their actual site address from the return address by analyzing the application binary.

instruction that causes the overflow can be queried. As an extension to that, Intel's PEBS (Precise Event Based Sampling) [60] not only provides precise location of the event in the instruction space, but also provides a way to access the register contents of the instruction that causes the event. Likewise, AMD's IBS (Instruction Based Sampling) supports reading the virtual address in the target register of retired `load/store` instruction together with its address. Similar sampling support is available on other microarchitectures such as Intel Pentium4/Itanium, IBM POWER5, Sun UltraSparc, etc.

Sniper samples memory accesses, i.e., `load(store)` events, to capture both the instruction and data addresses in the target register through the PMU based instruction sampling. Such information about each sample along with its timestamp is recorded in the trace files. Later, the trace simulator determines whether sampled instructions access a heap object. That is, if there exists a heap object that embraces the data address of the instruction when it executes, the simulator updates the *staleness* of the object based on its timestamp.

Currently, Sniper supports process- and thread-aware sampling with the help of Perfmon2 kernel interface [115] and ptrace system calls. Sniper intercepts process/thread creation requests through the ptrace hook, and creates a PMU context for each thread; the context contains appropriate PMU configurations including event types and sampling periods. Sniper then attaches the PMU context to the corresponding thread to be created.

On a context switch, the kernel reconfigures the PMU according to the attached PMU context, and enables the sampling of memory accesses. With this support, Sniper can monitor and save thread-level information thereby effectively dealing with multithreaded applications. During program execution, Sniper monitors the application's PMU transactions, i.e., memory access samples, and fetches the samples through the Perfmon2 kernel interface.

It is important to note that Sniper does not interrupt the target application execution at all. Thus, Sniper keeps track of the heap accesses without perturbing the original application execution. In particular, to prevent a majority of samples from falling into a synchronized pattern in some loops, Sniper leverages the sampling period randomization, i.e., adding a small randomized factor to the period.

5.3.2 Lightweight Heap Trace Generation

To track the *staleness* of the heap objects, Sniper has to be aware of the heap organization in terms of its allocation and deallocation during program execution. For this purpose, Sniper should record full traces of malloc/free and the related program context information. Even if the trace simulator takes over much of the heavy work such as tracking heap organization, Sniper still needs to minimize the overhead of the trace generation.

Unfortunately, the trace occupies a large amount of memory space to store the tag data of each heap object which includes its allocation/deallocation/last-access sites, heap organization information, e.g., the range information of an allocated object and freed address, etc[3]. To tackle the space overhead, Sniper buffers the tag data trace and flushes the buffer into a file when it is full. In this way, the memory consumption of the trace generation is bound to the buffer size.

Especially for a multithreaded application, Sniper should take care of contention to the buffer and the file from multiple threads. Of course, Sniper should guarantee that multiple threads write their trace correctly. One way to do that is relying on locking mechanism on the buffer and the file. However, this causes unacceptable performance degradation of the application due to the high synchronization overhead as the number of threads increases. Instead, Sniper allocates both a buffer and a file into each thread, thus they become thread-private. This makes the buffer

[3]The space for the *staleness* is not necessary at runtime because it is calculated offline by the trace simulator.

accesses lock-free and allows Sniper to use `fwrite_unlocked` for lockless file writing.

In particular, it is important for the trace simulator to have a synchronized view of traces from multiple threads. Note that Sniper achieves this with no additional cost, since it associates each of malloc/free/PMU traces with its timestamp in the first place for single-threaded applications.

5.3.3 Offline Trace Simulation

Sniper offloads the time- and space- consuming work of the *staleness* update for heap objects, which requires tracking heap organization and searching for the heap object accessed by a sampled instruction. To achieve this, Sniper leverages the lightweight heap trace generation and its offline trace simulator that takes over the heavy work. In particular, the simulator builds the tag directory based on recorded traces and performs expensive tag searches to calculate the *staleness* offline.

Once the traces of malloc/free/PMU(memory access) are recorded at the end of program execution, all the trace files are fed into the simulator. An application binary is also fed into the simulator for its binary analyzer to extract an actual allocation site address, i.e., the instruction address of `call` to malloc/new, based on their return address in the stack trace; the same thing is with deallocation and last-access sites.

Then, the simulator first merges the traces in the files and sorts them by the timestamp of each trace, which gives the simulator a time-synchronized view of all the traces, using MapReduce [36]. While the simulator is running, it decodes each trace in turn and performs appropriate actions according to the decoded results.

To keep track of heap organization, Sniper models each heap object with the start and end addresses of the object in the tag, and maintains a tag directory to

manage the tags. When the simulator processes a malloc (free) trace, the corresponding tag is created/inserted (removed) in the directory. For a memory access trace, the simulator determines if the address of the access corresponds to one of heap objects. This involves a search for the corresponding tag whose range (the start of the tagged object $\sim$ its end address) embraces the queried address in the tag directory[4]. If the search succeeds, i.e., a heap object access, the *staleness* of the resulting tag for the object is calculated and recorded by the simulator. Since it tracks the heap organization, each access is correctly attributed to the corresponding heap object.

In summary, the simulator replays the program execution in terms of heap usages, updating the *staleness* of the allocated objects. That way Sniper catches every leak occurred during the execution that generated those traces being simulated.

5.3.4 Efficient Implementation of a Tag Directory for Fast Heap Organization Tracking

For every heap access, i.e., PMU sample, the trace simulator needs to search the tag directory for the corresponding heap object. Therefore, the search performance dictates the choice of a data structure for implementing the tag directory. At the same time, the data structure should be compact in case the trace file is huge. In this respect, a hash table is not appropriate due to its lack of range searching capability; even if it can mimic the range search by inserting every byte address of an heap object, it causes huge space overhead. Another candidate for the directory implementation might be an interval tree that supports the range search. Unfortunately, their implementation is too heavyweight requiring O(NlogN) construction time, where N is the number of stored objects, to find all the intervals for a query.

[4]For the fast range search, we implements the directory using the specially modified red-black tree whose asymptotic complexities remains the same, i.e., O(logN) time. Appendix provides more deatil on it.

We use the following insight to propose a better data structure; a malloc guarantees that no two allocated objects overlap with the each other, thus there can be only one interval for a query in the tag directory. With that in mind, Sniper exploits a new data structure called *single interval tree*. This is a compromise between an interval search tree and a binary search tree. The idea is that visiting right child requires checking if the range of the current tag node, i.e., the start address to the end address, embraces the query address. Algorithm 3 shows the details of the range search in pseudo code.

Require: $data\_address$ from a sampled $load$ instruction
  $Link\_type\&\ x \leftarrow getRoot()$;
  while x is not null **do**
    if $data\_address < x.start$ **then**
      $x \leftarrow x.left\_child$
    else if $data\_address < x.end$ **then**
      return $iterator(x)$ // search succeeded
    else
      $x \leftarrow x.right\_child$
    end if
  end while
  return $end(x)$ // search failed

Algorithm 3: Search Operation in Single Interval Trees

Figure 31 shows an example of the single interval tree where each node represents a tag. The tags only show the range information in the figure. As an example, for a query address of 970, searched is the tag node with a range of [900, 990] in the tree, since the query address belongs to the range. To implement the single interval tree, this work modifies a Red-Black tree, a self-balancing binary search tree. Note that all other operations of a Red-Black tree do not need any modification. Consequently, the asymptotic complexities of the single interval tree remains the same as those of the Red-Black tree, i.e., O(logN) time for *insert/delete/search* operations and O(N) space. With the help of the single interval tree, Sniper can efficiently simulate huge traces of heavily multithreaded applications.

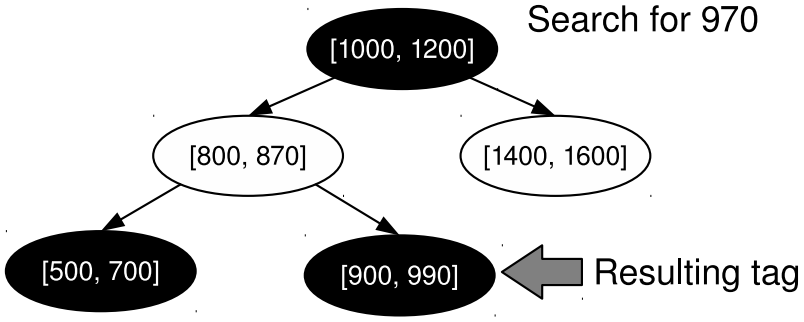


Figure 31: An example of a single interval tree based on a Red-Black tree. Numbers show the address range.

5.3.5 Systematic and Automated Leak Identification Using Anomaly Detection

Once the trace simulation finishes, Sniper is ready to report leaking objects based on the *staleness*. An important issue is how to determine the $threshold_{staleness}$. It is very important to precisely determine the threshold, since it directly impacts the number of false positives and negatives.

To the best of our knowledge, no prior work deals with this issue, thus users are left to set it properly. Unfortunately, it is indeed difficult, costly, and error-prone for users to set the threshold correctly. Upon any change, such a high cost of threshold determination will have to be paid anew. In particular, the threshold should be different across applications, and there is no one-size-fit-all solution even in the same application.

With that in mind, this work leverages a statistical anomaly detection, i.e., Sniper views leaks as anomalies. That is based on a couple of observations; (1) the *staleness* of a leaking object is very high compared to normal objects allocated in the same site, which is the basic philosophy of *staleness*-based approaches. (2) the number of leaks is a lot smaller than that of normal objects, which is true because production software should undergo a number of extensive testing procedures from its creation to the release. In fact, large software companies such as Google has already adopted the test-driven application development [103]. Without passing

various test cases, developers cannot submit even a single line of change to code repository. That way naive leaks in applications are likely to be detected before its production release.

5.3.5.1 *Anomaly Detection with Adjusted Boxplots*

Sniper transforms the problem of leak detection into that of anomaly detection for univariate data set which is comprised of the *staleness* of objects. The issue here is that most of anomaly detection techniques assumes underlying distribution, e.g., *boxplot* approach works best for normal distribution as other approaches favor it [139].

However, the leak detection problem does not follow normal distribution. Recall that leaks are not relatively many whereas normal objects are dominant, and the *stalenesses* of leaking objects is very large compared to that of normal objects. Thus, the distribution of *stalenesses* data tends to be right-skewed, i.e. having a long tail in the positive direction, which can paralyze the anomaly detection capability of a naive approach.

That leads to a different approach, i.e., Sniper leverages *adjusted boxplots* [139] for the anomaly detection. In contrast to the original *boxplot* that classifies all points outside the interval of $[Q_1 - 1.5\ IQR; Q_3 + 1.5\ IQR]$, where Q_1 and Q_3 are 1st and 3rd quartiles respectively and IQR is $Q_3 - Q_1$, as *potential* anomalies, the *adjusted boxplot* shifts the interval with the consideration of how the underlying distribution of the data set is skewed. For the systematic leak determination, Sniper sets the $threshold_{staleness}$ as the upper bound of the *adjusted boxplot* defined as;

$$Threshold = \begin{cases} Q_3 + 3.0e^{3\ MC}\ IQR & MC \geq 0 \\ Q_3 + 3.0e^{4\ MC}\ IQR & MC < 0 \end{cases}$$

where *medcouple* (MC), i.e., a *robust* measure of the skewness of underlying distribution, is defined as;

$$MC = \underset{x_i \leq Q_2 \leq x_j}{\text{med}} h(x_i, x_j)$$

where Q_2 is the sample median and for all $x_i \neq x_j$. That is, *MC* is the median of the kernel function (h) results where h is given by;

$$h(x_i, x_j) = \frac{(x_j - Q_2) - (Q_2 - x_i)}{x_j - x_i}$$

More details of *adjusted boxplots* can be found in [139].

5.3.5.2 The Granularity of the Anomaly Detection

This section describes leak detection schemes that differ in the scope of the anomaly detection, e.g., an entire application/allocation site.

Local Detection: Sniper can apply the anomaly detection for each allocation site, which is called local detection. This scheme has potential to achieve higher accuracy, since it performs allocation-site-specific (context-sensitive) leak detection. Even when the inequality 12 does not hold for entire objects, the scheme can still detect leaks with no false positive/negative. I.e., by narrowing down the scope of leak detection to those objects created in the same site, the inequality 12 is likely to hold.

However, the local scheme can be misleading depending on the state of an allocation site. They might occur for a couple of reasons; (1) insufficient amount of sample data; if some site has a few objects, e.g., < 10 objects, in which case no statistical method works. (2) similarity of sample data; even with the abundant amount of sample data, *stalenesses* of the objects could have not much difference, in which case even humans cannot detect any anomaly. E.g, it would be the case where every object created in one site is all leaking, or no object is leaking.

Global Detection: To deal with the problems, Sniper can perform the anomaly detection for entire objects within the application, which is called global detection.

Note that the global scheme still performs the application-tailored leak detection but not in the allocation-site-specific way. When the local scheme fails to detect leaks due to its high threshold ($local\_threshold_{staleness}$), the global scheme would be a good alternative. E.g., when those objects created in the same site are all leaking, the global scheme can still detect them in that its threshold ($global\_threshold_{staleness}$) is likely to be smaller than their stalenesses.

Then the question is how to make a correct decision to pick the right detection scheme for each case. By doing that, Sniper can take advantage of the synergy between the local/global schemes thereby achieving higher accuracy. On the contrary, an incorrect decision translates to false positives/negatives. With that in mind, this work designs Sniper's hybrid detection scheme.

Hybrid Detection: Sniper performs the local detection for each site in the first place. The idea is that Sniper respects the leak report of the allocation-site-specific detection scheme. For only those allocation sites that report no anomaly (leak), does Sniper consult the global detection scheme. For the local and global schemes to generate a different result, the staleness spectrum of the objects in the site has to be overlapped with the interval of the two thresholds of the both schemes. I.e., the candidate sites for the hybrid detection is defined as

$$\begin{aligned} candidate\_sites = \{site | site \in \mathcal{S}, global\_threshold_{staleness} < \\ \max_{o \in site} staleness(o) < local\_threshold_{staleness}(site)\} \end{aligned} \tag{13}$$

where $\mathcal{S}$ is a set of allocation sites in an application.

For each candidate site, the hybrid scheme simply uses the global scheme assuming that the site's objects are leaking. The intuition behind the heuristic is two-fold; (1) it follows the philosophy of the original *staleness*-based leak detection [28], i.e., highly-stale objects are likely leaking. (2) Sniper must not miss leaks. Otherwise, it loses its worth as a leak detection tool.

However, the heuristic might end up with false positives in case the assumption

is wrong. Note that this is rather a limitation of *staleness*-based leak detection, i.e., it is possible to incorrectly blame the objects that are highly *stale* but that do not actually leak. As an example, even if GUI objects might not be accessed for a long time after their creation, they should not be reported as leaks [147].

To avoid such unnecessary false positives, Sniper focuses on users' expectation for a leak detection tool. In general, users are interested in the critical leak that impacts overall memory consumption. That is, they would not care about a leak which rarely affects memory consumption, even though it is highly *stale*.

With that in mind, Sniper selectively applies the heuristic according to how much stale objects contribute to total memory consumption, i.e., the hybrid scheme switches to the global one only for the following sites;

$$\begin{aligned} hybrid\_sites = \{site | site \in candidate\_sites, \\ \sum_{obj \in site} size(obj) \quad > \quad \theta \sum_{obj \in allocated\_set} size(obj)\} \end{aligned} \tag{14}$$

where *allocated_set* is a set of the objects that have been allocated but not yet freed at time t_{report}. That is, if it turns out that the stale objects detected by the global scheme do not contribute that much, the hybrid scheme remains at the local scheme. Note that θ is a configurable parameter which takes into account the application's SLA and the QoS requirement. This work sets the value of θ to 0.1% in the experiments.

5.3.6 Robustness to False Positives due to Sampling

Since Sniper leverages instruction sampling to update the *staleness* of the accessed heap object, it could miss some memory access. Such a uncaught access to heap objects causes Sniper to overestimate their *staleness*. In a sense, Sniper might falsely report them as leaks, thus causing false positives.

It is important to note that for frequently accessed objects, the sampling does

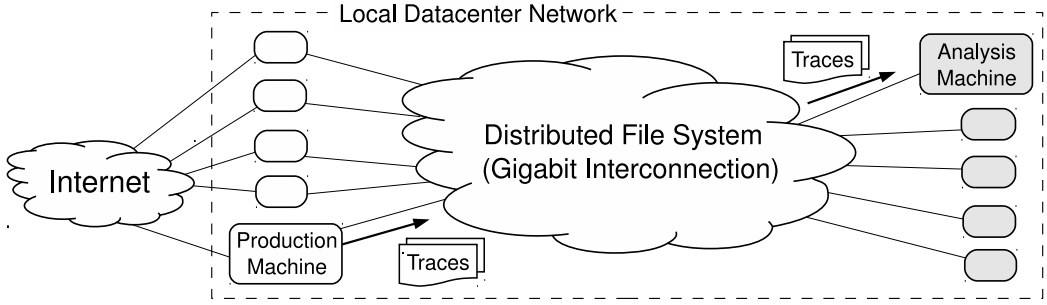


Figure 32: The datacenter environment

not have a significant impact on the false positives[5]. The reason is that often times leaks become manifest after long-running execution. I.e., it is practically impossible to generate false positives against frequently accessed objects for that long time. The real possibility of false positives due to the sampling comes from those objects that are sporadically accessed. E.g., if the last access to the objects with a long life time is not sampled, Sniper ends up overestimating the *staleness* thereby falsely reporting them as leaks.

In particular, Sniper turns out to be accurate even when the sampling frequency is low (see Section 5.4.5). That is because even if *staleness* gets overestimated due to the low sampling rate, Sniper's anomaly detection adapts itself to the underlying sample distribution. I.e., Sniper adjusts the *threshold* appropriately according to the resulting *staleness* distribution. Thus, Sniper can effectively prevent unsampled objects from being falsely blamed as leaks.

5.3.7 Discussion

Trace Size/Simulation Time: Without any optimization, the largest trace file we evaluated was ≈7 GB, and its simulation took ≈20 minutes including the time spent sorting the trace with MapReduce [36]. Table 8 summerizes both the trace size and the simulation time of those applications whose trace simulation takes

[5]In [28], Chilimbi and Hauswirth reported the same phenomenon in their execution-path-biased sampling technique.

Table 8: The trace size and the simulation time

Benchmark	Trace Size	Simulation Time
omnetpp	6832.5 MB	18.5 Min
dealII	3585.4 MB	9.3 Min
xalancbmk	3118.4 MB	8.6 Min

more than 5 minutes. For other SPEC2006 applications, the trace size is mostly less than 512 MB while the trace simulation takes a few minutes.

One possible optimization to reduce the trace size (simulation time) is periodically processing partial trace files during program execution. Once simulation outputs are generated, most of the trace files can be deleted. Those malloc traces having the corresponding free traces can be deleted too. However, for incremental *staleness* update, any information necessary to track the heap organization should be maintained. For efficiency, the partial trace files can be transmitted to other available machines in a pipelined way for the remote simulation.

Note that many datacenter applications have already collected various traces for a monitoring purpose. Thus, dedicated analysis machines often exist in the datacenter to process the log and trace data of production machines, which is true for Google's datacenter [112]. Sniper can thus leverage such machines to enable the remote simulation. Figure 32 shows the datacenter environment. In particular, both production and analysis machines share a distributed file system, e.g., Google File System [51], which is connected to a separate a gigabit network. Thus, writing a trace file rarely affects the QoS of the application which is serviced using another network, e.g., Internet; the production machines are equipped with two NICs for each separated network in order to keep the overhead minimal.

Limitation with Virtualization: The target of Sniper is non-virtualized datacenters where high-performance is a critical issue. E.g., almost all Google's production applications including Bigtable [24] run on a non-virtualized cluster node in the datacenter for performance reasons [94, 95]. Since the proposed PMU-based

technique does not assume a virtual machine (VM), Sniper is not directly applicable to VMs in its current form.

However, this is not a fundamental obstacle for Sniper to be used on virtualized datacenters. In the VM environment, PMU is shared among processes on different VMs as well as on the same VM. Therefore, PMU virtualization is a key to avoid mixing memory access samples of different processes. Recently, operating system researchers have come up with a framework for the PMU virtualization [105]. Currently, KVM (Kernel-based Virtual Machine) has already supported Intel's architectural PMU [4]. Thus, we expect that the support for other features of PMU to be available soon.

5.4 Evaluation

In order to demonstrate the effectiveness of Sniper, we implemented it in C++ as a shared library on a Linux operating system. To access PMU, we leverage Perfmon2 kernel interface [48]. This section first analyzes the time and space overhead of Sniper for both single- and multi-threaded benchmark suites with their largest input available. We run the benchmark applications five times using the largest input available, and show the median result for the applications; a geometric mean is used to calculate every average. Then, thie section presents a performance analysis with commercial datacenter workloads. And then, it analyzes the accuracy of Sniper using synthetic memory leak injection, and presents a sensitivity analysis to sampling period.

Finally, it describes a case study of using Sniper to detect real-world memory leaks in several open-source applications vulnerable to malicious denial-of-service attacks. All experiments were performed on a Linux machine with two Intel Nehalem-based quad-core Xeon processors (i.e., 8 cores total in two sockets) with 32GB of memory. In particular, trace files are written to a distributed file

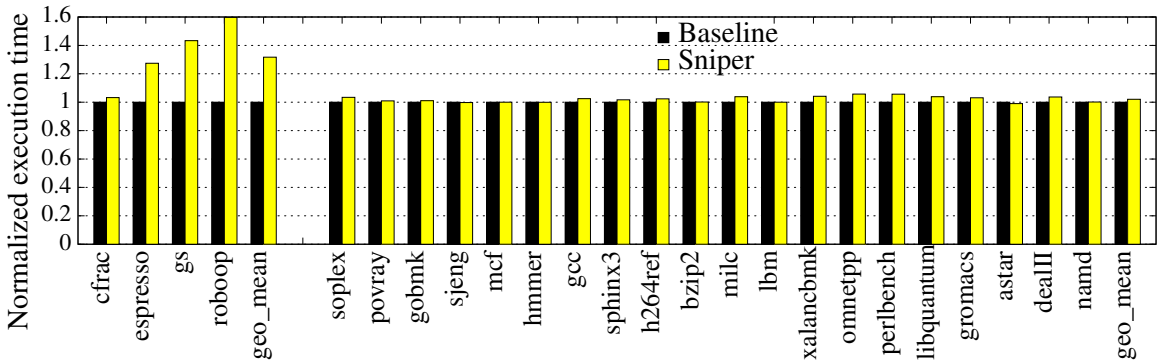


Figure 33: Execution time of SPEC2006/allocation-intensive benchmarks

system. This configuration is similar to one type of cluster nodes in commercial datacenters. Except for the sensitivity analysis, Sniper runs with a sampling period of 100.

5.4.1 Analysis with Sequential Applications

This section analyzes the time and memory overhead of Sniper for C/C++ applications from SPEC2006 benchmark suite. In addition, we measure the overhead for several allocation-intensive applications, since they were used in the most recent work [106].

5.4.1.1 Runtime Overhead of Serial Benchmarks

Figure 33 summarizes the execution time overhead incurred by Sniper in serial applications including both SPEC2006 and allocation-intensive benchmarks. In the figure, the dark bars correspond to a baseline execution time without Sniper, while light bars to the execution time with Sniper enabled, which is normalized to the baseline time.

For most of the SPEC2006 applications, Sniper's overhead is negligible (<1–3%) except for *xalancbmk* (4%), *perlbench* (5%), and *omnetpp* (6%). Note that prior work [106] omitted the applications but 483.xalancbmk in its evaluation, e.g., it failed to execute *omnetpp* due to its overhead. For *xalancbmk*, the prior work causes

huge overhead (almost 10x slowdown), while Sniper's overhead is only 4%. Overall, the execution time overhead incurred by Sniper is 3% on average for the SPEC2006 applications.

In addition, we measure the overhead for several allocation-intensive applications, since they were used in the most recent work [106]. For these applications, Sniper causes relatively significant overhead (3%–59%) despite its lightweight heap trace generation. However, such overhead is encouraging in that the applications spend a considerable amount of the entire execution time for memory allocation/deallocation. In fact, prior work [106] causes much more overhead (50%–100%) for the same applications. On average, Sniper's execution time overhead is 31% for the allocation-intensive applications.

Sniper can lead to performance degradation for three reasons. First, those applications are allocation- and deallocation-intensive, thus Sniper perturbs the application execution for a moment in order to store the meta data necessary to leave malloc (free) trace for each malloc (free) invocation. Second, they create many small heap objects, thus the meta data can become much larger than the the original size of the objects. As a result, the applications can cause more traffic to caches and TLBs. Finally, such many allocations/deallocation requests quickly make the trace buffers full. Thus, the file write operation to flush the buffers also occurs relatively frequently.

5.4.1.2 Memory Overhead of Serial Benchmarks

To evaluate Sniper's space overhead for sequential applications, This experiment measures the memory consumption while Sniper is running. Again, we ran all C/C++ applications from SPEC2006 benchmark suite and allocation-intensive applications. Figure 34 summarizes the memory space overhead incurred by Sniper across the benchmark applications. The chart configuration remains the same as

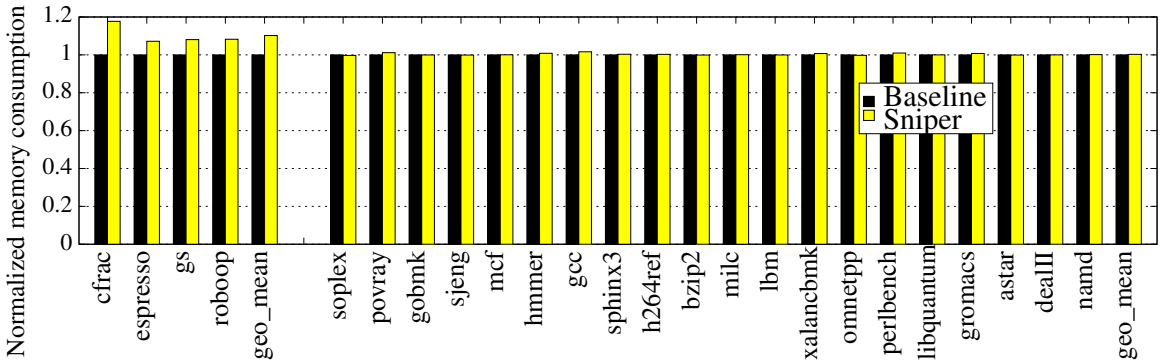


Figure 34: Memory overhead of SPEC2006/allocation-intensive benchmarks

before. The memory consumption was measured by taking multiple memory snapshots during program execution and computing their average. It is based on counting the number of occupied physical pages to calculate the total memory consumption by both the original application and Sniper. This means that the overhead incurred by Sniper is probably overestimated in that the pages could not be fully occupied due to fragmentation or malicious heap allocation/deallocation patterns.

It is important to note that there is no increase in application's heap size, since Sniper does not change the underlying memory allocator unlike prior work [106]. The only source of Sniper's space overhead comes from the trace buffers necessary to keep the meta data of each heap object. That is, the memory space overhead is bound to the buffer size; recall that Sniper buffers the malloc/free/PMU traces and flush them into files when the buffers become full. On average, the memory space overhead incurred by Sniper is 0.4% for the SPEC2006 applications.

For allocation-intensive applications, Sniper's overhead varies (7%–17%) depending on the applications. The reason why the overhead is high compared to SPEC2006 applications is mainly due to their memory allocation/deallocation pattern. They repeatedly allocate many small objects and deallocate them shortly. In fact, their original memory consumption is very small ($<$ 2MB), thus the trace buffers look relatively larger in these applications. On average, Sniper's memory

space overhead is 10% for the allocation-intensive applications.

5.4.2 Analysis with Multithreaded Applications

This section analyzes the time and memory space overhead of Sniper for PARSEC parallel benchmark suite.

5.4.2.1 Runtime Overhead of Parallel Benchmarks

To evaluate the execution time overhead for parallel applications, we chose PARSEC benchmark suite whose applications are heavily multithreaded and written in C/C++ [11]. Since the PARSEC applications run in parallel on multiple cores, this section focuses on Sniper's influence on the scalability of the original applications. Figure 35 represents how Sniper affects the scalability of the multithreaded applications as the number of threads increases. The solid line corresponds to a speedup of a baseline execution without Sniper, while the dashed line to a speedup of the execution with Sniper enabled. Overall, Sniper does not hurt the scalability of the applications. The main reason for this is that Sniper lets multiple threads have thread-private buffers and file pointers to dump the buffers for efficient trace collection. In this way, Sniper cannot only reduce contention to the buffers, but also can exploit lockless file operations. On average, when the number of threads used is 1, 2, 4, and 8, Sniper's execution time overhead is 3.3%, 3.8%, 4.3%, and 4.8%, respectively.

5.4.2.2 Memory Overhead of Parallel Benchmarks

This experiment measures the memory consumption of PARSEC applications while Sniper is running. Figure 36 summarizes the memory space overhead incurred by Sniper across the applications, when the number of threads used is eight. In the figure, the dark bars correspond to a baseline memory consumption without Sniper, while light bars to the memory consumption with Sniper enabled, which

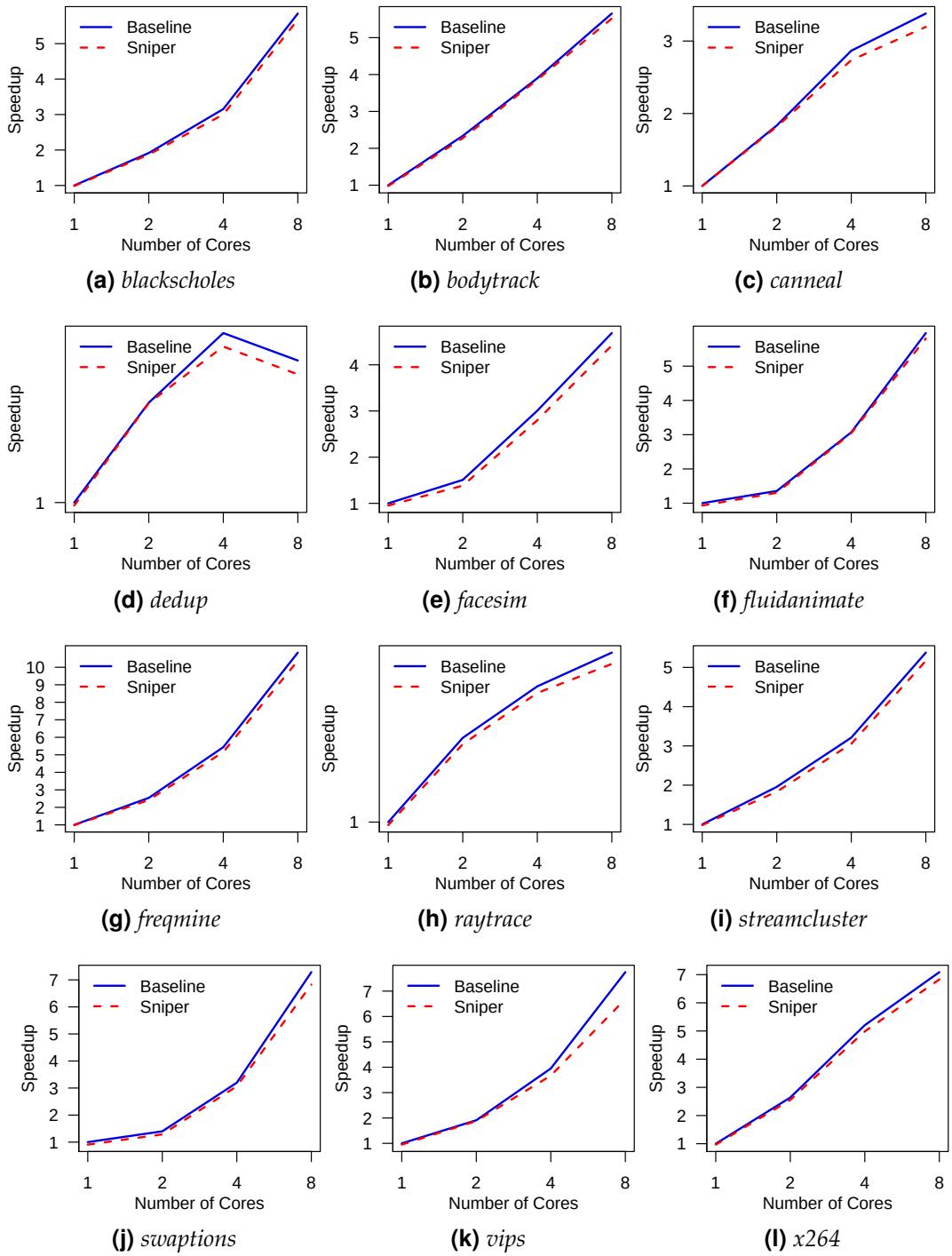


Figure 35: Scalability of PARSEC parallel benchmarks

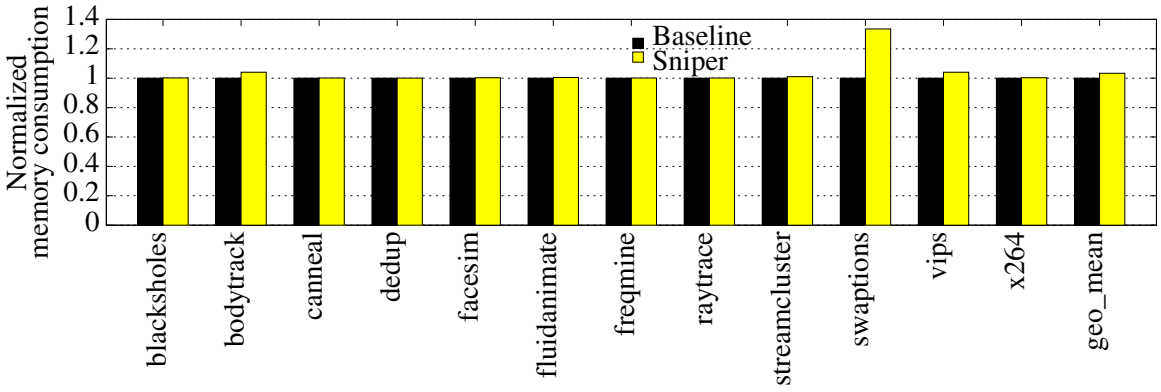


Figure 36: Memory space overhead of PARSEC parallel benchmarks

Table 9: Commercial datacenter benchmarks

Benchmark	Lines	Description	Overhead
A	$\approx$ 1 M	Web search engine	3.7 (%)
B	$\approx$ 1 M	Ads search engine	3.2 (%)
C	$\approx$ 1 M	Application server	1.9 (%)
D	$\approx$ 1 M	Protocol buffers	3.3 (%)
E	$\approx$ 1 M	Panoramic image stiching	1.4 (%)
F	$\approx$ 200 K	Openssl encryption	0.9 (%)
G	$\approx$ 100 K	(De)compression	1.1 (%)

is normalized to the baseline memory consumption. For most of the applications, the space overhead is negligible (<1–4%) except for *swaptions* (33%). In particular, the memory footprint of the *swaptions* benchmark is very small (< 3MB). Thus, Sniper's space overhead, i.e., the size of trace buffers for each thread, ends up looking much larger in the application. On average, Sniper's memory space overhead is 3.3% for the PARSEC parallel applications.

5.4.3 Analysis with Commercial Datacenter Workloads

We also evaluated Sniper with a set of large C++ benchmarks used currently at Google's datacenter. They span a wide range of applications from the web search engine to the image stitching for panoramic streetview in maps. Except for benchmark D and E, all the others have a very large code base with million lines of source code. Table 9 briefly introduces each benchmark application. The fourth

column of the table entries shows the performance overhead when Sniper is running with each application. In particular, the benchmark D is comprised of over 20 test applications whose performance is aggregated to calculate the average performance. I.e., the performance of this benchmark shown in the table is a geometric mean. Overall, the performance overhead incurred by Sniper is not that significant (roughly 1 to 4%) in those commercial datacenter applications. Thus, Sniper does not hurt the QoS of the datacenter applications, which is defined as the execution time or the throughput.

5.4.4 Accuracy Analysis with Leak Injection

To evaluate the leak detection accuracy of Sniper, there is a need of a good set of applications containing various memory leaks. with which the detection accuracy is measured and compared. However, there is no such standard applications to the best of our knowledge. This work therefore creates leak benchmarks stress-tested with the synthetic leak injection. We inject two types of leaks, i.e, dynamic and static leaks, into C/C++ SPEC2006 applications[6]. In real-world applications, memory leaks often times manifest only in certain program contexts (e.g. specific procedure calling sequence or malicious user input patterns). To model this kind of memory leaks (called *dynamic leaks*), we first run the original SPEC2006 applications with Sniper to collect the `free` traces, and then randomly remove 10% of deallocations from the traces.

On the other hand, leaks sometimes occur irrespective of the contexts (called *static leaks*), e.g., every object created in a single allocation site is leaking. Even if such leaks are relatively rare in deployed software due to extensive in-house testing, they become a serious problem whenever they occur. That is because every created object gets lost and never reaches any deallocation site, thereby leading to

[6]The experiment omits *mcf*, *sjeng*, *lbm* since these applications allocate very few objects, i.e., their results tend to be misleading.

memory bloat. To model this scenario, we work first pick an allocation site which is responsible for closest to 10% of entire allocations, and then remove deallocations of the objects created from the site. As metrics to evaluate the leak detection accuracy, we use *precision, recall, F-measure* that are commonly used to measure the quality of classifiers in the information retrieval community. Intuitively, high precision leads to less false positives (falsely blamed leaks) while high recall to less false negative (undetected leaks), and the F-measure is the harmonic mean of precision and recall which focuses on the balance between the other two metrics.

To compare accuracy of different approaches, we test the *ad hoc* approach (i.e., using manual *threshold*) used in prior tools with Sniper approach that automatically selects the threshold using anomaly detection. Remember that due the lack of the systematic methodology, prior tools end up using a fixed threshold across applications. To model the *ad hoc* approach by selecting the appropriate threshold, we first try 20 candidates forming *arithmetic series* among which the smallest value results in no false negative while the largest in no false positive. Then, we select a value with the best F-measure as the threshold. Thus, the real *ad hoc* approach may perform worse than what we model here.

In particular, to quantify and verify effectiveness of Sniper's heuristic for the hybrid anomaly detection (described in Section 5.3.5.2), we implement an ideal hybrid approach based on oracle information. That is, the ideal approach (called *Ideal Hybrid*) always selects the best between the global and local leak detection schemes.

Figure 37 compares precision/recall of different leak detection approaches; (1) *Ad-Hoc*: the manual approach of prior work, (2) *Sniper-Hybrid*: Sniper's leak identification based on the hybrid anomaly detection, (3) *Ideal-Hybrid*: the ideal version

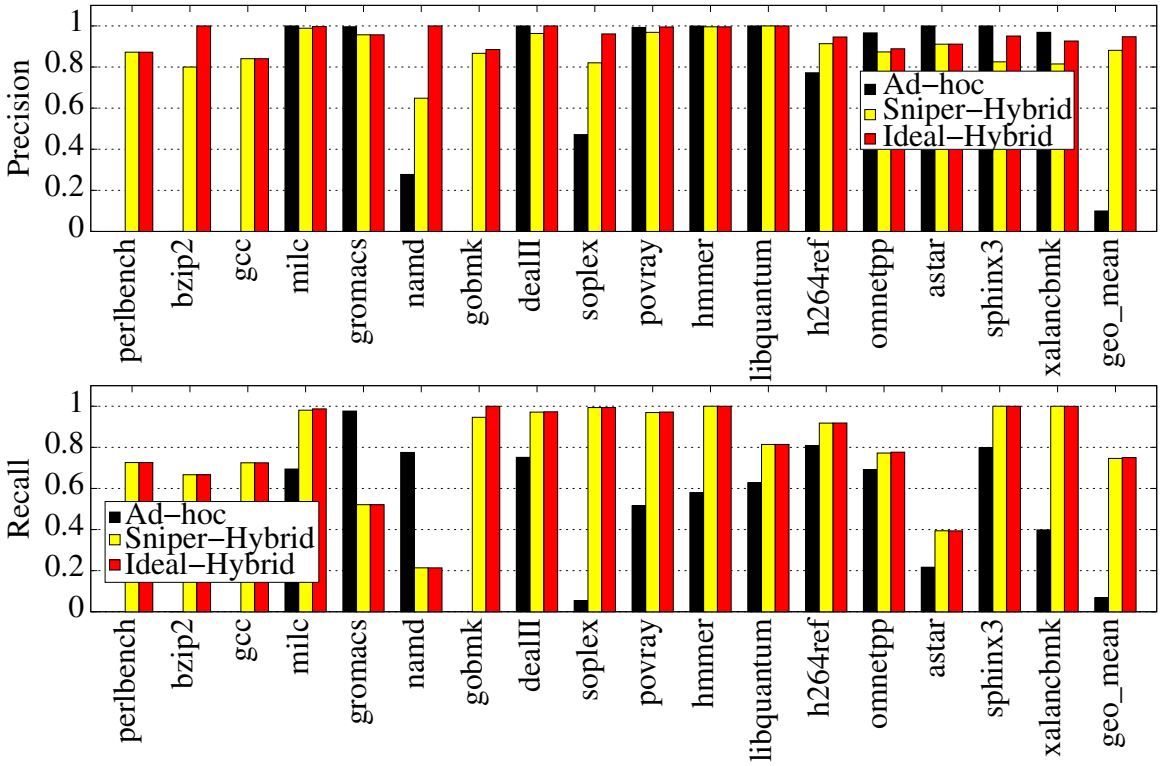


Figure 37: Precision and recall of different leak detection approaches. Sniper's accuracy is shown in the second bar, i.e., *Sniper-Hybrid*.

of Sniper based on the oracle information. For most applications, *Hybrid* outperforms *Ad-Hoc*. This is due to Sniper's application-tailored leak identification strategy. *Sniper-Hybrid* works comparably in *namd*, *omnetpp*, *sphinx3*. and it is less accurate than *Ad-Hoc* only in *gromacs*. In *perlbench*, *bzip2*, *gcc* and *gobmk*, *Ad-Hoc* does not work at all and the results translates to its low average of precision/recall. On the contrary, *Sniper-Hybrid* can fit itself into each problem instance by examining underlying *staleness* distributions and never has a case where it fails to detect all the presence of leaks.

In *gromacs* and *astar*, *Sniper-Hybrid* fails to detect *static leaks*, which are supposed to be caught by the global anomaly detection scheme (See Section 5.3.5.2), thus resulting in low recall. However, it turns out that the global detection could not detect the *static leaks* either. The reason for this is that in *gromacs* 31% of allocations end up leaks due to the leak injection. Because of such a large number, the

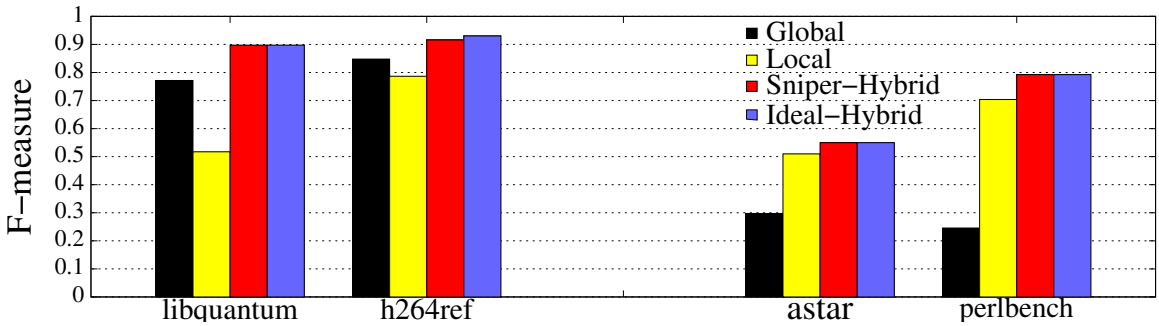


Figure 38: The impact of the hybrid anomaly detection. Sniper's F-measure is shown in the third bar, i.e., *Sniper-Hybrid*.

leaks does not look like anomalies to the global detection, i.e., it cannot distinguish leaks from innocent objects. Similarly, *astar* has *static leaks* too, thus *Sniper-Hybrid* suffers from the same problem. As an exception, it achieves low precision and recall in *namd*. That is because the *stalenesses* of innocent objects and leaks in *namd* are so severely overlapped that it cannot accurately separate leaks even with its allocation-site specific local detection scheme.

Note that *Sniper-Hybrid* is near-optimal for most applications, i.e., it is as accurate as *Ideal-Hybrid*; it turns out that most of the time, Sniper's heuristic for the hybrid anomaly detection correctly selects the best between the local and global detection schemes. There are four exceptions (*namd*, *soplex*, *sphinx3*, *xalancbmk*). That is because *Sniper-Hybrid* is either too conservative or too aggressive for them. I.e, for *namd*, *Sniper-Hybrid* is too conservative due to the severe overlap in the application while it is too aggressive for the rest of them. Overall, *Sniper-Hybrid* is very accurate; its precision and recall are 0.88 and 0.75, respectively, and the resulting F-measure is 0.80.

One reason for the high accuracy of *Sniper-Hybrid* is that its heuristic for the hybrid anomaly detection successfully selects the best between the local and global detection schemes. Figure 38 shows the F-measure of each scheme, and highlights how *Sniper-Hybrid* behaves when either the local detection scheme (*Local*) or the global scheme (*Global*) works better than the other. In *libquantum* and *h264ref*,

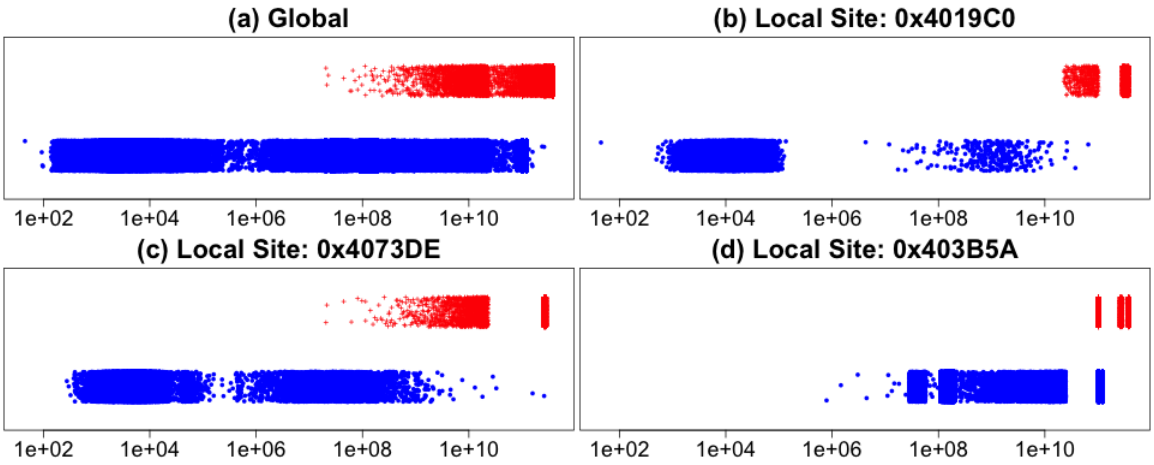


Figure 39: Stalenesses spectrum of objects in *perlbench* shown in a log scale

Global outperforms *Local*. That is because these applications have relatively many *static leaks* for which *Local* is destined to fail. Here, even the local detection reports no anomaly, *Sniper-Hybrid* catches the leaks by correctly switching to the global detection.

On the other hand, *Local* outperforms *Global* for *astar* and *perlbench*. In particular, they have relatively many *dynamic leaks* which are supposed to be caught by the local anomaly detection scheme (Section 5.3.5.2). As a result, the applications show considerable overlap in the *stalenesses* of *dynamic leaks* and innocent objects, which prevents the global scheme from detecting the leaks. That is why *Global* achieves the low accuracy for the applications. In contrast, the local detection scheme can solve this problem with the help of its allocation-site-based partitioning of objects.

Figure 39 shows four 1-D scatter plots demonstrating the benefit of such partitioning. Each point in the plot represents *staleness* of an object. Leaks are plotted in the upper part of a plot while innocent objects are plotted in the lower part. As shown in Figure 39(a), before partitioning, there are huge overlap between *stalenesses* of leaks and innocent objects. I.e., here it is very difficult for *Global* to recognize the leaks an anomalies.

However, when the objects are partitioned according to their allocation sites,

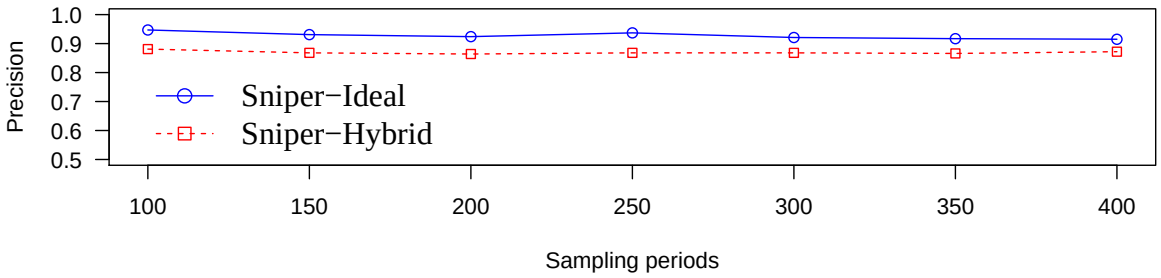


Figure 40: Impact of sampling period change on false positives (Precision)

the degree of overlap within each partition reduces a lot. Thus, those objects of each partition become much more amenable to the anomaly detection. To support this, the rest of 1-D scatter plots in Figure 39, i.e., (b), (c), and (d) show three different allocation sites after partitioning. Here, *Sniper-Hybrid* makes correct decision, i.e., adopting the local anomaly detection scheme.

Overall, *Sniper-Hybrid* is comparable to *Ideal-Hybrid* and thus performs better than both the local and global detection schemes. Comparing the impact of the local scheme only and the global scheme only, it is clear that they have a constructive effect in *Sniper-Hybrid* which is the combination of *Global* and *Local*. I.e., *Sniper-Hybrid* achieves better accuracy than either scheme of them can. Apart from that, the fact that *Sniper-Hybrid* achieves the optimal accuracy of *Ideal-Hybrid* supports that Sniper's hybrid leak identification is accurate and effective.

5.4.5 Sensitivity to Sampling Frequency

Since Sniper leverages the instruction sampling, an unsampled access to heap objects causes their *staleness* to be overestimated thus possibly leading to false positives. Sniper should be robust against such false positives to be useful in datacenters that might force the sampling period to be adjusted for the SLA satisfaction.

Figure 40 shows Sniper's average precision on sampling period changes. Here, the average precision is the geometric mean of the precisions of C/C++ SPEC2006

applications. Overall, the precision change of Sniper is not significant across different sampling periods. For example, when the period is 400, i.e., Sniper observes a single access out of 400 memory accesses seen by the PMU, the resulting precision (*Sniper-Hybrid*) is still high; 0.872.

This is because even if the *staleness* overestimation due to the coarse sampling is inevitable, Sniper's anomaly based leak identification adapts itself to the resulting sample distribution of *staleness*. In other words, the automatic anomaly detection adjusts the *threshold* appropriately to determine leaks. In particular, *Sniper-Hybrid* is comparable to *Ideal-Hybrid* across the sampling periods.

5.4.6 Case Study of Real-World Memory Leaks Vulnerable to Denial-of-Service Attacks

Squid is a web caching proxy [2] widely used in datacenters. It caches frequently-requested web pages and delivers the contents from its local cache upon request from many users (clients), thereby improving the response time and the network bandwidth. Squid has a memory leak which could potentially be used by malicious attackers to crash the program or cause some system failure, i.e., denial-of-service. The root-cause of the problem is that invalid HTTP requests with an empty URL trigger a control path in which the memory allocated to serve the request will not be deallocated. To reproduce the leak, we ran Squid for several hours requesting many valid web site addresses along with the problematic URL at a constant rate.

It turns out that Sniper successfully detected the memory leak with no false positive. In addition, Sniper found that based on its simulation outputs, every non-leaking object created for valid HTTP requests has the same free site, and that all the objects including leaks have the same allocation site. By simply checking the object counts and the free site, the user can further recognize that most of the objects are not leaked—e.g., 95%—and freed in a single location. Then, it would be

a natural reaction for the user to attempt to deallocate the rest 5% leaking objects in the same location where 'all the non-leaking objects' (95% of the entire objects) are deallocated, which is the solution for the leak problem of Squid.

Packet-o-matic is a multithreaded network packet analyzer used in the local datacenter network. It performs network forensics [1], thus reading network packets and logging various information about the network connections. It has a memory leak due to incorrect thread termination. When the application reads an input file (*pcap* capture file) that generates the network traffic, a new thread is created to process the file. The problem is that even if each thread is supposed to *join* at its termination, (i.e., specified as *joinable* in the *pthread_create*), there is a case where it does not execute *pthread_join*.

According to Sniper, the leaking objects are all allocated in the same function, i.e., *pthread_create*. That is, they are a sort of thread local resources, which should be returned to the system at the end of the thread execution; such unredeemed objects accumulate as the application reads more input files. Here, Sniper's information of the last_access to the objects can help the user find the appropriate location to put *pthread_join*. In fact, the exact joining point was in the end of the pthread worker function. In order to give users the context information, e.g., the allocation site, the pthread library were statically compiled in this experiment. The alternative is to instrument the dynamic loader (ld.so) so that it can leave the information on where the pthread library is loaded in memory [69]. In particular, Sniper generated no false positive for this application.

USIMM is an open source architecture simulator for memory scheduling [63]. It had severe leaks causing the simulator to eventually crash with an out-of-memory error when the simulation input is large. The root-cause is that memory requests already serviced are not deallocated even if they do not exist in the service queue any longer. Since there are billions of memory requests being scheduled in the

queue, the simulation can eventually eat up all the available memory in the system.

Sniper turns out to be very accurate, i.e., no false positive, in detecting the memory leak in USIMM. Note that in this case, the last-touch site information helps to figure out the cause of the leak. Sniper successfully reported that the site where the memory requests are serviced. For developers with full understanding of how the simulator schedules the requests with the queue, the site information motivates them to investigate the function of clearing the queue where `free` is supposed to exist to fix the leak.

5.5 Summary

Memory leak detection in datacenters is a critical step toward the QoS enforcement, the reliability enhancement, and the reduction of both the SLA violation and the operational cost. This work presents Sniper, an effective memory leak detection tool. Its runtime overhead is negligible (mostly $<$3%) and never increases the application's heap size. Sniper is also applicable to multithreaded applications without hurting the scalability. Thus, Sniper can be practically used in datacenters and observe real execution characteristics in production runs thereby effectively detecting memory leaks, which are inherently input- and environment-sensitive.

To the best of our knowledge, Sniper is the first to provide a systematic methodology for accurate leak identification. Sniper automatically determines the *staleness* threshold based on an anomaly detection. As a result, the leak identification is tailored not just for each application but for each allocation site as well, thus Sniper achieved an F-measure of 81% on average for 17 benchmarks stress-tested with various memory leaks. We believe that our statistical methodology improves the accuracy of other leak detection approaches that use different sampling techniques. In particular, Sniper is a transparent unlike prior tools; it does not change

application behaviors by modifying the executable or replacing the original memory allocator. The empirical evaluation demonstrates that Sniper is highly accurate in detecting critical memory leaks in real-world software.

CHAPTER VI

DATA STRUCTURE ACCELERATION

6.1 Introduction

Data structures are the main focus of performance engineering. They are one of the most critical aspects in determining the performance of many real-world applications. Indeed, it is not difficult to find situations where improved data structure usage can result in orders of magnitude improvement in application performance. E.g., Chung et al. report that they achieved more than 20x speedup by carefully optimizing 2-D table data structures used in their work[30], and there are many examples; matrix multiplication [141], information mining from large databases [7], and analyzing genetic data for patterns [54], just to name a few.

To this end, computer architects have investigated various hardware support to accelerate common data structures including trees and graphs [84, 135, 10, 100, 85, 38, 91] over the last several decades. Wu et al. [143] devised ADP (Abstract Datatype Processor), special hardware acceleration for hash tables and sparse vectors. To improve memory performance of priority queues, Chandra and Sinnen [23] implemented HardwarePQ based on the full shift-register architecture [100]. Recently, Bloom et al. [13] proposed a dedicated hardware logic system called HWDS (Hardware Data Structure) as well as an exception model to support large queues. All these efforts turn out to effectively improve the application performance by accelerating the hot code of the data structure manipulation.

However, any commodity processor has not yet supported such special hardware. Even if FPGA-based data structure accelerators might be available, the operating system and compiler support to leverage them must be addressed in the first

place for their pervasive adoption. In light of this, to improve data structure usage, this work takes a software-only approach called data structure offloading, simply DSO. It is inspired by previous helper threading approaches [68, 80, 151, 124, 43, 71, 88, 32, 19]. In particular, this work leverages a helper thread running on an idle core to offload expensive data structure operations of an application. That is, the helper thread executes the data structure code on the behalf of the application thread. While the former operates on the data structure, the latter can keep executing the rest of its code. Thus, the helper thread can take the cost of performing the expensive operations away from the application thread.

This work first recognizes a critical data structure and its operations in the profile run of the application. For effective communication and synchronization between the helper and application threads, we leverage Lamport's lockfree queue that does not require any hardware support [76]. I.e., the application pushes the argument of the operations to the lockfree queue for offloading them. Thus, data structure offloading effectively replaces the complex data structure operations with simple lockfree queue operations. Since both helper and application threads run in parallel, they need to be carefully synchronized for the data structure not to be corrupted. For program correctness, users are required to place the synchronization code, which is also realized using the lockfree queue, at the right place if necessary. However, in reality, users often end up generating redundant synchronization code in the presence of the complex control flow. With that in mind, this work presents a compiler algorithm to eliminate such redundant synchronization code automatically.

The empirical results demonstrate that data structure offloading (DSO) can achieve significant speedups for several applications that are inherently sequential and hard to parallelize. It delivers a significant speedup from 1.12 of 1.30 for data structure intensive real-world applications. Note that even if this work evaluates

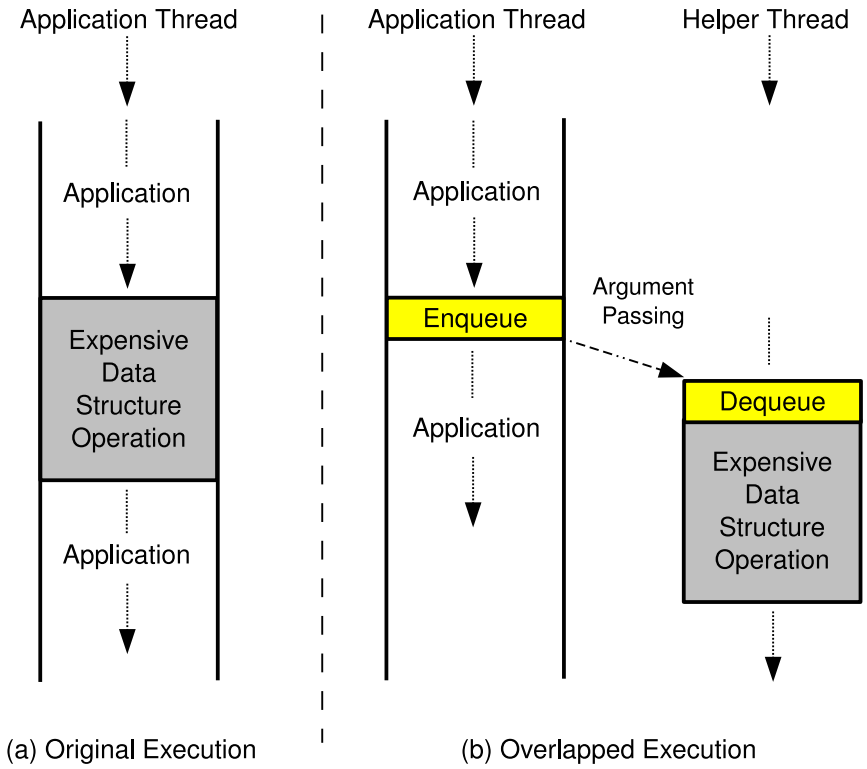


Figure 41: The idea of data structure offloading: (a) original sequential application execution versus (b) overlapped execution

DSO with sequential applications, the proposed techniques are directly applicable to parallel applications as long as idle cores are available to run helper threads. In general, this situation is often very common because many parallel applications are not fully scalable due to either resource contention or insufficient amount of parallelism [66, 82, 81, 108]. I.e., their best performance is achieved when not all the available cores are used.

6.2 Offloading Expensive Data Structure Operations

The main idea of data structure offloading (DSO) is to hide the cost of performing the expensive data structure operations of an application by offloading them. This is achieved by leveraging a helper thread which performs the operations on the behalf of the application thread. If the operations being offloaded are to `insert`

(`erase`) to (from) the data structure, the application thread can keep executing the rest of its code; we call them non-blocking data structure interfaces[1] while `find`-like operations are called blocking interfaces since the application needs to wait for what they return. To communicate the data if necessary, DSO leverages Lamport's single-producer/single-consumer concurrent lockfree queue [76]. E.g., to offload an `insert` interface, the application thread explicitly executes a `push` operation[2] to deposit the data argument being stored in the lockfree queue.

Right after that, it can execute the very next code without waiting for the helper thread to finish the offloaded work of `insert`. Accordingly, DSO allows the application to effectively replace the complex data structure operation with the much simpler lockfree queue operation, i.e., `push`. Figure 41 describes this situation, i.e., how the helper thread takes the cost of performing the expensive operation away from the application thread.

However, there is no point in offloading blocking interface, since the application anyway must wait for the completion of the interface due to the data dependence. E.g., while the helper thread executes *find* operation on the behalf of the application, it must wait for the return value of the operation. In this case, there is no performance benefit at all due the lack of the overlapped execution. More seriously, the overhead of executing lockfree operations might rather degrade the overall performance of the application.

Note that since both the threads run in parallel with each other, we must carefully deal with the case where the application thread accesses the data structure while the helper thread is still executing the offloaded operation. Otherwise, the data structure might be corrupted and behave incorrectly ending up with program

[1] This work uses the terms operation and interface interchangeably.
[2] `push` is a lockfree queue operation to enqueue a data item to the circular buffer of the queue.

failure. For the program correctness, DSO must guarantee the sequential semantics of the original application. That is, the original calling sequence of the interface functions of the data structure has to be preserved using explicit synchronization. E.g., if the application invocations `insert(1024)` and `find(1024)` in turn and offloads the former to the helper thread, then it must finish the offloaded operation before the application attempts to find the data, i.e., 1024.

In particular, there is no need to care about the order between offloaded operations due the FIFO nature of the lockfree queue. For instance, when the application offloads two consecutive invocations to `insert` with different data, their calling sequence are guaranteed to be the same with the help of the lockfree queue; the helper thread executes the offloaded operations in order with regard to the euqueuing order of the application. In this respect, the easiest strategy to guarantee the sequential semantics is offloading every interface of a data structure. Again, offloading even blocking interface is likely to degrade the overall performance of the application, thus that is not a right strategy.

With that in mind, we offload only non-blocking operations, thus only caring about invocations to non-offloaded interface functions of the data structure as candidates for the synchronization. As a result, every call site to such interface functions [3] becomes the synchronization point where the application must wait for the helper thread to finish all the previously offloaded operations completely. To achieve the synchronization, DSO leverages the lockfree queue again. By checking if the queue is empty, the application can determine if it can keep executing passing over the synchronization point. Thus, the synchronization is implemented by repeatedly checking the empty condition until the lockfree queue is fully drained.

[3] If other functions access the offloaded data structure, i.e., it is not encapsulated well, then their call sites are synchronization points too. However, such situation would be rare provided the application is developed in the object oriented philosophy.

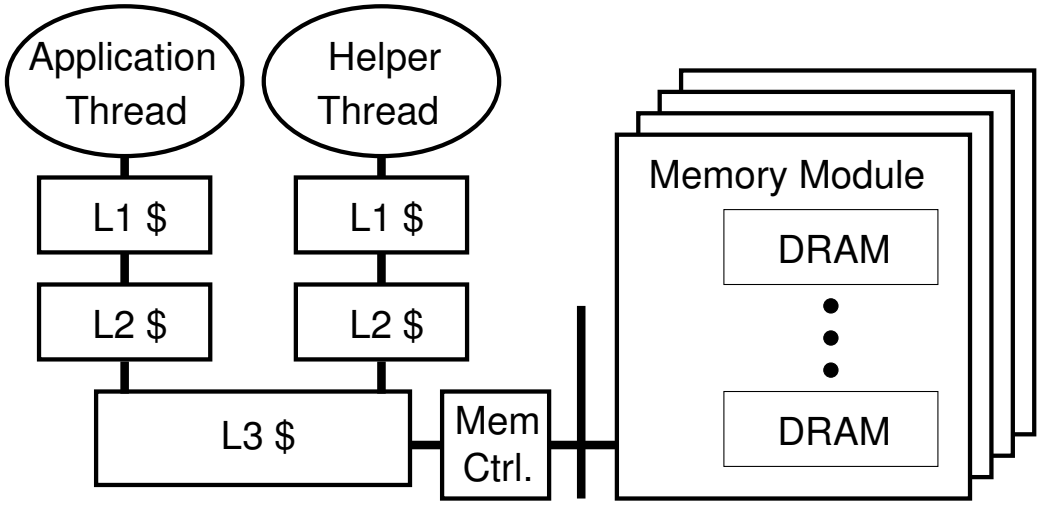


Figure 42: The target CMP system: Intel Nehalem microarchitecture

For this purpose, the application is supposed to explicitly execute a `sync` instruction which is a macro that performs the emptiness checks of the lockfree queue.

To a large extent, users often end up placing the `sync` instruction everywhere they think it is necessary. That is because users are always under the pressure of guaranteeing the correctness without program failure; what they want is the performance improvement not the segmentation fault. I.e., even when they are not completely sure if the `sync` is necessary at some program point, they tend to place the synchronization code there for safety reason. In reality, in the presence of the complex control flow, it becomes a very difficult task to precisely recognize the right synchronization point. To this end, users are likely to place redundant `sync` instructions along the complex control flow. Therefore, it is desired that the compiler automatically eliminates such redundant `sync` instructions.

Figure 42 describes the target processor architecture of DSO. To avoid unnecessary resource contention, it is important that the application and helper threads are run on different cores in the same chip of a standard CMP (chip multi-processor) system as showed in the figure. For this purpose, this work pins both the threads

to separate cores using a system call, i.e., `pthread_setaffinity_np`, at the beginning of program execution. Thus, they are never migrated to any other cores during program execution.

6.3 Compiler-Time Redundant Synchronization Elimination

Executing the `sync` instruction is the overhead of data structure offloading (DSO). Especially when the amount of offloaded work is not significant compared to the overhead, it would be difficult for DSO to improve the overall performance of the application. That is because the benefit from the overlapped execution of DSO is offset or dominated by the synchronization overhead.

This work founds out that depending on the complexity of application code, many of the `sync` instructions are often redundant, thus removing them does not violate the sequential semantics for the program correctness. In light of this, this work presents a global dataflow analysis to recognize such unnecessary `sync` instructions and to remove them automatically.

This section describes a compiler algorithm that statically eliminates the redundant `sync` instructions. In particular, it is assumed that an application can offload the operations of different data structure instances. This means that there can be multiple lockfree queue instances too, even if our previous example shows only one instance.

6.3.1 Local Redundant Synchronization Elimination

In a basic block, if the same `sync` instruction appears multiple times without any intervening `push(q, data)` instruction. Then, except for the first `sync` instruction, all the others are safely eliminated. Here, the `sync` removal decision is made without regard to the `data` argument of the `push` instruction, thus we omit it hereafter. For example, the instruction sequence is like `push(q); sync(q); find(q); sync(q); find(q)`, the last `sync` instruction is redundant thus can

be eliminated. To make the compiler analysis easier, we transform the `push(q)` instruction into a write to `q`, i.e., the instruction makes a new definition of `q`. Then for the decision making of the elimination alrogithm, the analysis can simply check if the same definition of `q` is used in both the `sync(q)` instructions. Note that if another intervening `push(q)` appears right after the first one, then the last `sync(q)` cannot be eliminated since its `q` has been re-defined. Thus, the analysis only needs the USE-DEF chain of `q` that is directly available on SSA (static single assignment) forms [6, 102].

6.3.2 Global Redundant Synchronization Elimination

Across basic blocks, the compiler analysis should be able to eliminate redundant `sync` instructions in the presence of complex control flow. To achieve this, this work leverages a global dataflow analysis. The main idea of the analysis remains the same, i.e., once a `sync(q)` instruction is made, it should be used until the program point where the `q` is re-defined. Only difference is that we need to propagate the `sync` information along the control flow. In the following, we introduce a set of definitions necessary for the analysis, and present the dataflow equations, and then describe an algorithm that eliminates redundant synchronization code based on the dataflow analysis results. First, *available sync* instruction is defined as below.

Definition 3 *A* `sync(q)` *is available at a given program point P if the following conditions are satisfied.*

- *the* `sync(q)` *has been performed on every path to P from the entry node of the CFG (control flow graph).*
- *the argument* `q` *has not changed since the last time it was computed on the paths to P.*

Similarly, *killed sync* instructions are defined as below.

$$
\begin{array}{rcl}
Avaiable\_Check\_In[B] & = & \bigcap_{(P \in Pred(B))} Avaiable\_Check\_Out[P] \\
Avaiable\_Check\_Out[B] & = & \{Avaiable\_Check\_In[B] - Kill[B]\} \cup Gen[B] \\
Avaiable\_Check\_In[B] & = & \begin{cases} \bigcap_{(P \in Pred(B))} \{Avaiable\_Check\_In[P] - Kill[P]\} \cup Gen[P] & \text{if } Pred(B) \neq \{\} \\ \{\} & \text{if } Pred(B) = \{\} \end{cases}
\end{array}
$$

Figure 43: The dataflow equations: *Available_Check_In* and *Available_Check_Out*

Definition 4 *A* `sync(q)` *is killed by the definition of* `q`*. I.e, only when it is written by* `push(q)`*, killed is the sync that uses the* `q`*.*

Let us then define *Gen* and *Kill* sets, respectively. For a given basic block *B*, the *Gen[B]* is defined as below.

Definition 5 *Gen[B] is comprised of the* `sync` *instructions of the basic block B that are not killed by any of later definitions, i.e.,* `push(q)` *in the block.*

Then for a given basic block *B*, the *Kill[B]* is defined as below.

Definition 6 *Kill[B] is comprised of the* `sync` *instructions outside the basic block B that are killed within the basic block B*

Figure 43 describes the dataflow equations based on the definitions above. It first shows the definitions of *Available_Check_In* and *Available_Check_Out*. Since each equation is defined in terms of the other, *Available_Check_In* can be derived by substituting the *Available_Check_Out* with its definition showed in the figure. In particular, if there is no preceding basic block, the resulting *Available_Check_In* should be an empty set. This additional condition prevents the *Available_Check_In* from being a universal set. I.e., without this special care, the dataflow analysis ends up propagating all the `sync` instructions along the control flow, which is problematic.

Finally, Algorithm 4 describes the overall process of the redundant synchronization elimination in pseudo code. Once the dataflow analysis equation is solved,

the algorithm iterates each basic block and evaluates its `sync` instructions consulting the analysis results, i.e., *Available_Check_In*. A `sync(q)` in a basic block, b, is redundant and thus can be eliminated if *Available_Check_In[b]* has the same `sync` instruction and it is *available* at the program point where the `sync(q)` appears. Algorithm 4 performs this by determining if there is no intervening `push(q)` between the basic block entry and the location of the `sync(q)` being currently evaluated, and by invoking $Eliminate($`sync(q)`$)$ if that is the case.

Algorithm 4: Complete Redundant Check Elimination with Available Check Dataflow Analysis

Require: Basic_Blocks: a set of basic blocks
Require: Checks[]: a set of `sync` instructions in a basic block
  for all $b \in$ Basic_Blocks **do**
    Compute $Gen[b]$ and $Kill[b]$
  end for

  // Solving the dataflow analysis
  while $Available\_Check\_In[\,]$ changes **do**
    for all $b \in$ Basic_Blocks **do**
      $Available\_Check\_In[b] = \bigcap_{(P \in Pred(b))} \{Avaiable\_Check\_In[P] - Kill[P]\} \cup Gen[P]$
    end for
  end while

  // Eliminating redundant synchronization
  for all $b \in$ Basic_Blocks **do**
    for all `sync(q)` $\in$ Checks[b] **do**
      if `sync(q)` $\in Available\_Check\_In[b]$ **and** $\nexists$ `push(q)` between $EntryPoint(b)$ and $Point($`sync(q)`$)$ **then**
        $Eliminate($`sync(q)`$)$
      end if
    end for
  end for

6.4 Experimental Evaluation

In order to demonstrate the effectiveness of data structure offloading (DSO), we implemented the proposed compiler algorithm as part of the LLVM toolset [77].

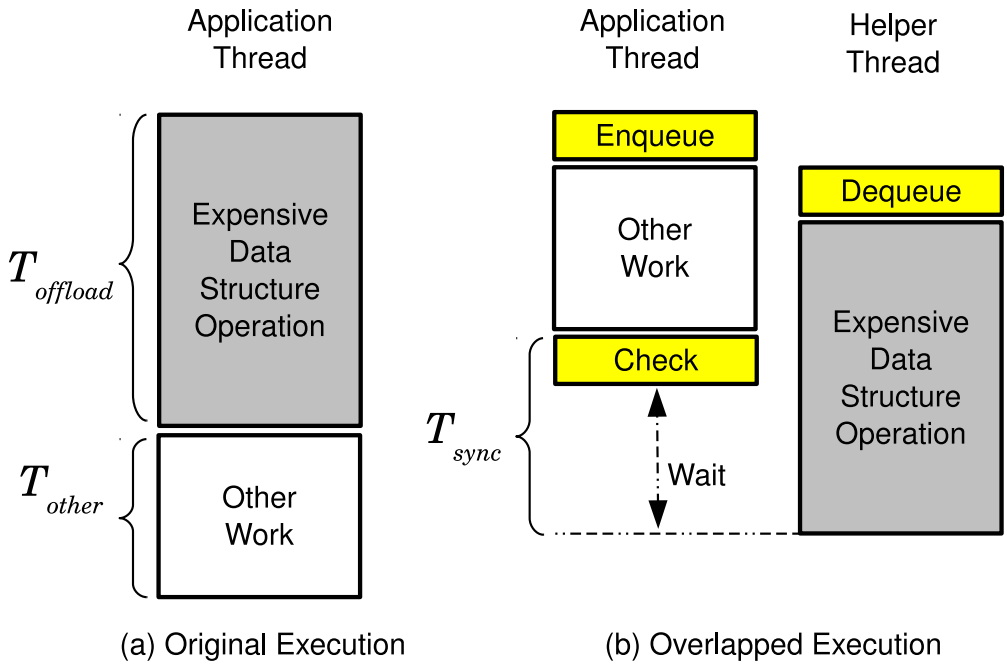


Figure 44: Insufficient overlapped execution and its performance impact

The Lamport's lockfree queue was implemented as a C++ template. All the experiments were performed on a Linux-based system with Intel Xeon E5520 2.26 GHz 8-Core CPU and 24GB of RAM. Table 10 shows the system configuration in detail. This section first presents the performance characterization of DSO to understand the impact of overlapped execution of the application and helper threads, and then presents two case studies.

Table 10: Target system specification

CPU	Intel Xeon quad-core E5520 2.26GHz
Microarchitecture	Nehalem
Caches	64 KB L1 cache per core, 256 KB L2 cache per core, 8 MB L3 unified
Memory / Disk	24 GB SDRAM / 1 TB HDD
Operating System	64 bit Linux with a kernel 2.6.30
Compiler	GCC 4.4 with libc/libstdc++ 4.4.0

6.4.1 Performance Characterization

In this section, we evaluate and analyze the effects of both the lockfree queue overhead and the amount of overlapped execution on the performance of data structure offloading (DSO). On the one hand, DSO can improve the overall performance of the application by allowing the application to replace its expensive data structure operation with much simpler lockfree queue operation. Intuitively, the longer operation the application offloads, the better performance it can achieve. On the other hand, the overall performance of DSO is affected by the ratio of the time that the application spends executing the expensive operations to the rest of the application execution time. That is because the amount of overlapped execution of the application and helper threads, which leads to the time saved, varies depending on the ratio.

To obtain a more precise idea of the extent and nature of the overlapped execution, we evaluate the speedup of a microbenchmark when the time ratio is varied. It has a hot loop whose body inserts a couple of integers to a priority queue implemented by a binary search tree and erase one of the existing data in the queue, and performs some random work, and then checks the size of the queue. Before entering the loop, the microbenchmark populates the priority queue with a fixed number of random integer values. To enable the overlapped execution, the first two priority queue operations are wrapped with a macro function, and it is offloaded to the helper thread. Since the end of the loop body has an access to the priority queue in order to get its current size, the application thread has to wait for the helper thread to finish the offloaded operation. In this experiment, we vary the amount of the random work in the middle of the loop body, thus changing the ratio of the original time spent in the offloaded operation to the rest of the application execution time.

Based on the speedup measurement, it turns out that the overall performance

initially improves as the ratio increases, and it reaches the peak speedup of 1.38, and then it starts to decrease even if the ratio keeps increasing. The reason for that is because when the offloaded work is larger than the rest of application work, DSO cannot fully overlap the executions of the application and helper threads. Figure 44 describes how this happens. Here, due to the insufficient amount of the overlapped execution, the helper thread cannot fully hide the time spent performing the expensive data structure operation. The figure also demonstrates that the lockfree queue operations, e.g., `enqueue` , `dequeue`, can affect the overall performance of the application.

With that in mind, we introduce a simple speedup model to understand the performance potential of DSO. T_{enq} and T_{deq} are enqueue and dequeue times shown in Figure 44, while T_{check} is the time spent checking whether or not the lockfree queue is empty. Then, the speedup can be calculated as below.

$$Speedup = \frac{T_{offload} + T_{other}}{T_{enq} + T_{other} + T_{sync}}$$

It is assumed that the queuing delay is included in the T_{deq}. In particular, T_{sync} varies depending on the ratio of $T_{offload}$ to T_{other}, and it is defined as below.

$$T_{sync} = \begin{cases} T_{check} & \text{if } T_{other} >= T_{deq} + T_{offload} \\ T_{deq} + T_{offload} - T_{other} & \text{otherwise} \end{cases}$$

When there is a sufficient amount of the overlapped execution, T_{sync} is just the time spent checking the emptiness of the lockfree queue, thus T_{sync} equals to T_{check}. Otherwise, i.e., within the time of T_{other} the helper thread cannot finish the offloaded operation, T_{sync} becomes the time the application thread must wait until the the operation is finished as shown in Figure 44. By substituting T_{sync}, the speedup will

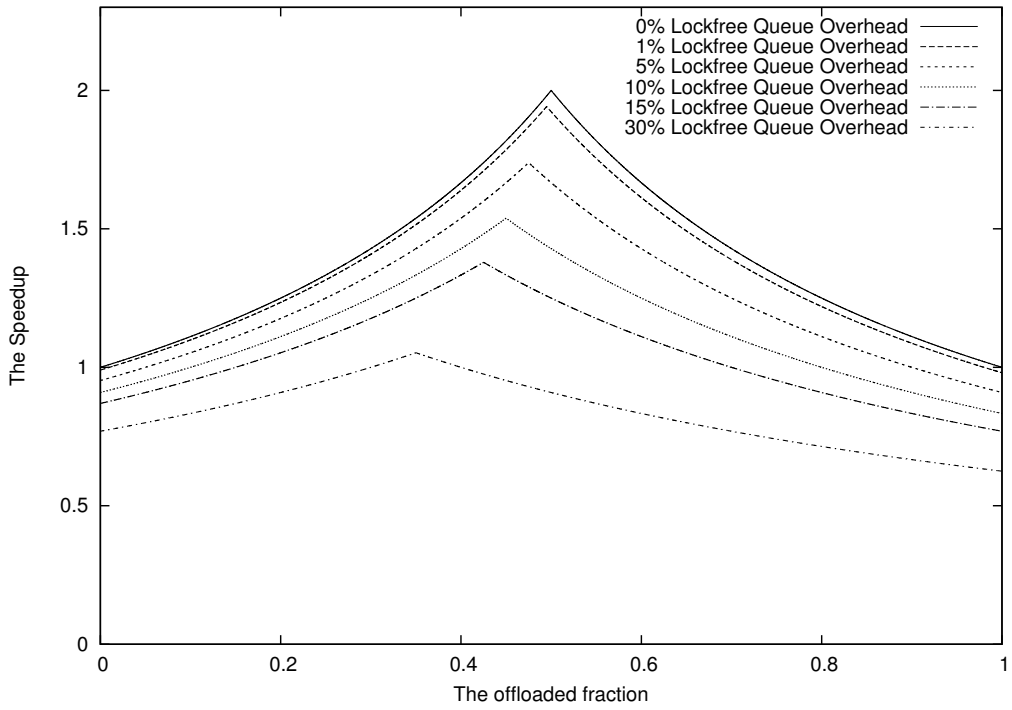


Figure 45: The offloading performance

be:

$$Speedup = \begin{cases} \frac{T_{offload}+T_{other}}{T_{enq}+T_{other}+T_{check}} & \text{if } T_{other} >= T_{deq} + T_{offload} \\ \\ \frac{T_{offload}+T_{other}}{T_{enq}+T_{deq}+T_{offload}} & \text{otherwise} \end{cases}$$

In the speedup formula above, T_{check} is small compared to the other two lockfree queue operations that have similar costs. For simplicity, we ignore T_{check}, assume that T_{enq} and T_{deq} takes the same time, and represent it with $F_{overhead} \in [0, 1]$ in the formula. In addition, if the offloaded fraction of the application execution time is $F_{offload} \in [0, 1]$, then the speedup will be:

$$Speedup = \begin{cases} \frac{1}{1-F_{offload}+F_{overhead}} & \text{if } F_{offload} <= \frac{1-F_{overhead}}{2} \\ \\ \frac{1}{F_{offload}+F_{overhead}\times 2} & \text{otherwise} \end{cases}$$

Figure 45 shows how the speedup changes when $F_{offload}$ increases. In particular, $F_{overhead}$ is also varied to evaluate the impact of the lockfree queue overhead on

the speedup. The best performance is achieved when the fraction of the original application that are not offloaded is the same as a sum of the $F_{offload}$ + $F_{overhead}$. If the $F_{offload}$ becomes larger than the equilibrium, the speedup starts to decrease. Especially, as the overhead of lockfree queue operations increases, this equilibrium is reached more rapidly. For example, when $F_{overhead}$ is 0.15, i.e., T_{deq} is 15% of the original execution time of the application, the maximum speedup is made at the point where $F_{offload}$ is 0.425.

There are two lessons from the observation. First, the selection of a candidate data structure operation for offloading should take into account how far the synchronization needs to be made. For example, if the `sync` instruction has to immediately follow the offloading point where `push` instruction appears, there will not be a sufficient amount of the overlapped execution DSO can achieve. In this respect, if the compiler can schedule the code in a way that a distance between the `push` and `sync` instructions is maximized. Second, the execution time of the candidate operation being offloaded needs to be sufficiently long enough. Otherwise, the resulting offloading performance is likely to be offset by the overhead of the lockfree queue operations, e.g., the target application repeatedly executes quick data structure operations. In such a case, the hardware supports, that can accelerate the lockfree queue operations, are essential to preserve the offloading performance [84].

6.4.2 Xalancbmk

Xalancbmk is an XSLT processor that performs an XML to HTML transformations. It takes as inputs an XML document and an XSLT stylesheet file that contains detailed instructions for the transformation. The program maintains a string cache comprised of two levels, *m_busyList* and *m_availableList*, `vectors`. When a string is freed in *XalanDOMStringCache::release*, it moves the string to the *m_availableList*

provided it is found in the *m_busyList*. This makes the `find` operation on the *m_busyList* one of hottest functions in the application.

However, the `find` function is not a non-blocking function meaning that it should not be offloaded to the helper thread. That leads us to investigate its caller function, i.e., *XalanDOMStringCache::release*, in turn to find another offloading candidate. Initially, the caller function has a return value. However, our compiler verified the return value is not used by leveraging an aggressive interprocedural analysis. Thus, we could safely offload the *XalanDOMStringCache::release* to the helper thread. Our compiler analysis successfully eliminated redundant *sync* instructions that were mislaid in the first place. To evaluate the performance improvement of DSO, we ran the program without any special option, i.e., the source document validation is not enabled. When a reference input available in the SPEC2006 benchmark suite, DSO achieved a speedup of 1.28.

6.4.3 SSSP

This program solves the single-source shortest path problem using Dijkstra's algorithm. It has a doubly nested loop and maintains a priority queue built on top of using `set` in C++ standard template library (STL) whose implementation is a red-black tree. In the outer loop, the program picks the node that currently has the minimum distance estimate by consulting the priority queue. In the mean time, the inner loop iterates the edge list of the minimum-distance node updating the distance estimate of the neighbors if necessary. During the iteration of the inner loop, the priority queue is repeatedly accessed to accommodate the neighbor whose distance estimate is updated with a smaller value. I.e., the `set::insert` operation of the priority queue becomes a hot function. Note that the cost of `set::insert` is quite expensive since it iterates the red-black tree possibly causing cache misses as well as performs multiple tree rotations if necessary. To this end, we offloaded the

Table 11: The speedup results of SSSP for each input graph

Input	Edge density	Number of vertices	Speedup
A	$m = 8n$	1048576	1.12
B	$m = 16n$	65536	1.20
C	$m = 32n$	1048576	1.24
D	$m = 64n$	65536	1.26
E	$m = 128n$	1048576	1.30

`set::insert` operation to the helper thread. Again, our compiler analysis successfully eliminated all the redundant *sync* instructions mislaid in the first place.

As an input of the program, we used the most common graph type called $G_{n,m}$ used as an input into Dijkstra's algorithm. In the graph, n and m describe the number of vertices and the number of edges, respectively. Thus, they represent the edge density of the input graph. In particular, the DIMACS web page provides the script that generates arbitrary graphs with different n and m values [41]. The script randomly connects different vertices for a graph with a specific edge density. Here, we evaluated the performance improvement of DSO with several input graphs of a different edge density. The table 11 represents the edge density of each input graph and the resulting speedup of DSO for the input graph. As shown in the table, when the edge density of a graph increases, the resulting speedup increases too. That is because when the graph becomes denser, i.e., having many more neighbors, the number of `set::insert` operations also increases in the original application. Thus, that contributes to the increase in the offloaded fraction of the application execution time.

6.5 Summary

Data structure offloading (DSO) is a promising technique to accelerate sequential applications that are hard to parallelize. By offloading a time-consuming data

structure operation to a helper thread running on an idle core, the application performance can be improved with a minimal effort to place necessary synchronization code. DSO achieves effective communication and synchronization by leveraging a concurrent lockfree queue without any hardware support. In particular, our compiler analysis automatically eliminates redundant synchronization code misplaced by users. We realized DSO on commodity processors, and demonstrated that it can achieve significant speedups for data structure intensive real-world applications.

CHAPTER VII

RELATED WORK

This chapter classifies the related research into data structure detection, data structure selection, and memory leak detection. Finally, it presents the research related to data structure acceleration.

7.1 Related Research on Data Structure Detection

There is a long history of work on detecting how data is organized in programs. Shape analysis (e.g., [52, 116, 140]) is among the most well known of these efforts. The goal of shape analysis is to statically prove externally provided properties of data structures, e.g., that a list is always sorted or that a graph is acyclic. Despite significant recent advances in the area [75, 149], shape analysis is provably undecidable and thus necessarily conservative.

Related to shape analysis are dynamic techniques that observe running applications in an attempt to identify properties of data structures [39]. These properties can then be used to automatically detect bugs, optimize applications [29, 111], repair data structures online, or improve many software engineering tasks [40]. While this type of analysis is not sound, it can detect properties outside the scope of static analysis and has proven very useful in practice.

This previous work statically proved or dynamically enforced data structure consistency properties in order to find bugs or optimize applications. The work here takes a different approach, where we assume the data structure *is* consistent (or mostly consistent), and use the consistency properties to identify how the data structure operates. *We are leveraging consistency properties to synthesize high-level*

semantics about data structures in the program.

The reverse-engineering community has also done work similar to this effort [5, 110]. These prior works use a variety of static, dynamic, and hybrid techniques to detect interaction between objects in order to reconstruct high-level design patterns in the software architecture. In this paper we are interested not just in the design patterns, but also in identifying the function of the structures identified.

The four works most similar to ours are by Raman et al. [111], Dekker et al. [37], Cozzie et al. [33], and Jump et al. [65], Raman's work introduced the notion of using a graph to represent how data structures are dynamically arranged in memory, and utilized that graph to perform optimizations beyond what is possible with conservative points-to or shape analysis. Raman's work differs from this work in that it was not concerned with identifying interface functions or determining exactly what data structure corresponds to the graph. Additionally, we extend their definition of a memory graph to better facilitate data structure identification.

Dekker's work on data structure identification is exactly in line with what we attempt to accomplish in this paper. The idea in Dekker's work was to use the program parse tree to identify patterns that represent equivalent implementations of data structures. Our work is more general, though, because (1) the DDT analysis is dynamic and thus less conservative, (2) DDT does not require source code access, and (3) DDT does not rely on the ability to prove that two implementations are equivalent at the parse tree level. DDT uses program invariants of interface functions to identify equivalent implementations, instead of a parse tree. This is a fundamentally new approach to identifying what data structures are used in applications.

Cozzie's work presented a different approach to recognizing data structures: using machine learning to analyze raw data in memory with the goal of matching

groups of similar data. Essentially, Cozzie's approach is to reconstruct the memory graph during execution and match graphs that look similar, grouping them into types without necessarily delineating the functionality. Instead this paper proactively constructs the memory graph during allocations, combines that with information about interface functions, and matches the result against a predefined library. Given the same application as input, Cozzie's work may output "two data structures of type A, and one of type B," whereas DDT would output "two red-black trees and one doubly-linked list." The take away is that DDT collects more information to provide a more informative result, but requires a predefined library to match against and more time to analyze the application. Cozzie's approach is clearly better suited for applications such as malware detection, where analysis speed is important and information on data structure similarity is enough to provide a probable match against known malware. Our approach is more useful for applications such as performance engineering where more details on the implementation are needed to intelligently decide when alternative data structures may be advantageous.

Jump and McKinley propose ShapeUp for Java to recognize the shape of recursive data structures during program execution [65]. This analysis constructs a *class field summary graph* (CFSG) and checks in- and out-degree invariants of the CFSG whenever garbage collection occurs. They show that, even though the invariant check is performed periodically at garbage collection time, artificially injected bugs in microbenchmark are detected successfully.

However, this method is inherently incapable of detecting bugs which are temporarily hidden between garbage collection executions. The reality is that bugs show changing behavior thus it is unrealistic to assume that bugs appear at garbage collection times. Similarly, data structures also change, e.g., a binary tree can look like a linked-list according to particular data insertion and deletion patterns. If this

happens just before at a garbage collection time, ShapeUp could not detect the binary tree. In fact, it is quite possible for data structures to lose their defined shape due to data removal from them.

In addition, since ShapeUp relies on type information and garbage collection, it is also restricted to languages which provide managed runtimes and whose binaries maintain sufficient type information. Thus, ShapeUp is not applicable, for example, to an application binary compiled for C/C++. In addition, since ShapeUp does not consider invariants of a data structure's internal representation, i.e., the data invariants, their work cannot differentiate between those data structures whose shapes are very similar or exactly same. Again, the data invariants are necessary to discern a binary search tree from binary trees.

7.2 Related Research on Data Structure Selection

Selecting the best data structure implementation is often a problem ignored by developers; they simply rely on library developers to choose a good implementation for the average case and accept the results. This leaves significant room for improvement. When developers do select specific implementations, they typically rely on asymptotic analysis, even though it can often lead to incorrect decisions in real-world applications. As pointed out, asymptotic analysis was always intended to be used in algorithmic selection and not in data structure selection/tuning.

Several researchers have previously investigated the problem of data structure selection in various contexts [118, 119, 120, 87, 67]. Jung and Clark propose a dynamic analysis that can automatically identify data structures and their interface functions. They showed that the resulting information, e.g., how the functions interact with the data structures, is very useful for data structure selection [67]. Other researchers suggests language level supports for data structure selection. For example, in high-level programming languages, such as SETL, it is impossible to

select data structure implementations; all data structures are specified as abstract data types, and the compiler must determine the implementation [118]. Work in this area focused on using only static analysis for data structure selection [119]. While the raised abstraction level of these languages did help productivity, the performance of these tools was generally worse than hand-selected implementations.

The Chameleon [120] and Perflint [87] projects are the most similar to Brainy. Chameleon and Perflint instrument Java and C++ applications, respectively, to collect runtime statistics on behaviors such as interface function calls. Additionally, Chameleon collects heap-related information from the garbage collector. These statistics are then fed into hand-constructed diagnostics to determine if the data structures should be changed. Both Chameleon and Perflint showed impressive space and performance improvements for real-world benchmarks. In particular, Brainy considers memory bloat as Chameleon does. Recall that the application generator varies the number of data elements in a data structure as well as the size of each element, thus the generator can create applications suffering from memory bloat. Brainy extends those prior works by 1) using machine learning to automatically construct more accurate models, instead of relying on hand-construction, and 2) incorporating hardware performance counters into the analysis, thus providing greater accuracy. In particular, unlike Chameleon, Brainy is not restricted to languages that have managed runtime features such as garbage collection.

The prior works have three problems. First, they require many models for each data structure replacement. For example, if M data structures can be replaced with N alternative data structures, the prior works require total M x N models. Note that modeling the execution of alternative data structures depends on the original data structure. On the contrary, Brainy needs only M models, thus the instrumentation overhead can be greatly reduced.

Second, modeling the accurate execution of the alternative data structure is inherently difficult and sometimes impossible. For example, in a `vector-to-set` data structure replacement, it is very difficult to know how many data elements are accessed for a `find` operation in the alternative data structure (`set`) just by instrumenting the code of the original data structure (`vector`); that requires exactly tracking data insertion and deletion, operation order, orderedness of data values, search patterns, and so on. This subtlety of modeling the execution behavior of the alternative data structure forces the prior works to rely on asymptotic analysis and average case. However, such approximation is likely to generate inaccurate models. Thus we conclude that if the resulting models are inaccurate, why pay the cost of heavy instrumentation code for M x N models?

In this work, rather than modeling the execution of the alternative data structure, Brainy's machine learning-based model tries to answer the question, *'what alternative data structure is desirable when the original data structure behaves in a certain way?'* That is, Brainy focuses on modeling how the original data structure is executed to identify the relation between the execution location and the alternative data structures suited for the role. Consequently, Brainy reduces the number of models required compared to the prior works.

The last problem is about using hardware features, which are important as shown in Section 4.5.1. Unlike software features such as the number of function calls and their costs, it is almost impossible to model hardware features of the alternative data structure. For example, the number of mispredictions of conditional branches in the original data structure has no causal relation to the number of mispredictions in the alternative data structure. Thus, the prior works cannot effectively exploit hardware features while Brainy's machine learning-based model can. Again, the hardware features are critical for effective data structure selection.

In a different perspective, a body of work has been done to address inefficient

use of data structures in terms of memory bloat [96, 97, 148, 146]. In [97], Mitchell and Sevitsky suggest a systematic approach to detect those data structures that end up with unnecessary memory (bloat). They introduces a new notion, *Health*, that analyzes how the memory space of a data structure is organized and used; and they present judgement schemes based on the notion to determine the inefficiency of the data structure use [148]. Xu and Rountev also present static and dynamic tools that detect inefficiently used data structures to avoid. They first identify interface functions (e.g, ADD/GET) of a data structure using static analysis. Then the static or dynamic tools analyze how these interface functions are called during the data structure execution.

There are key differences between these prior works and Brainy. First, they target Java and rely on virtual machine support. Again, Brainy is not restricted to languages that have managed runtime features. Second, they can deal with only case of the bloat-caused inefficiency of data structures. In C/C++, bloat is less of a concern than in Java where garbage collection is very important. Brainy can deal with many more cases of data structure inefficiency. Finally, they do not select a data structure, i.e., they just show if a data structure is inefficient in terms of bloat. In contrast, Brainy does provide a solution for inefficient usage of a data structure by selecting an alternative data structure.

7.3 Related Research on Memory Leak Detection

There is a large body of existing research addressing the issue of memory leaks. Memory leaks are of two kinds: (1) *unreachable* memory, i.e., program cannot access it, and (2) *dead* memory, i.e., it is reachable, but the program will never use it again, thus it is not *live*. The *unreachable* leaks can be effectively addressed by garbage collection [15] and static analysis [57, 144, 26, 70] techniques. However, the *dead*

leaks are much more tricky, since it is in general undecidable to determine if certain memory will not be accessed in the remainder of the program execution. This difficulty leads to the advent of many dynamic analysis techniques that leverage *staleness* of allocated objects to determine leaks [109, 18, 128, 147, 16, 17, 31, 20, 145]. Since the focus of Sniper is to detect the *dead* leaks without a managed runtime, This section limits the discussion to dynamic techniques that can detect *dead* leaks without a managed runtime. conducted to date for C/C++ program. In particular, those tools [104, 93, 31, 20] based on full-tracing not only cause a huge slowdown (10x–300x) due to the instruction instrumentation, but also occupy a considerable memory space, e.g., half of the entire memory capacity in the case of shadow memory. For this reason, we do not consider them for further detail.

Path-Biased Sampling: Chilimbi and Hauswirth [28] were the first who proposed the *staleness* based leak detection in their pioneering system called SWAT. The *staleness* update relies on code instrumentation. To reduce the overhead, SWAT uses path-biased sampling in tracking heap accesses. It samples each program path at a different rate; the sampling rate is in inverse proportion to the execution frequency. That way SWAT can reduce the overhead, since instructions on a hot path rarely get sampled. However, the sampling can result in overestimating the staleness of the objects in hot paths, thus leading to false positives. Thus, the effort to reduce the runtime overhead may end up undermining the quality of the leaks detection.

Uniform Sampling versus Path-Biased Sampling: One might wonder which sampling is better. The path-biased sampling was invented to reduce overheads at the expense of hot-path's precision. Thus, only if the cold-path hypothesis holds, the path-biased sampling is more precise than uniform sampling. However, there is doubt about the generality of the hypothesis. As [106] pointed out, it does not

hold for many cases, e.g., data structure operations, where the path-biased sampling generates many false positives.

Even if the hypothesis holds, Sniper is still robust against false positives. That is, Sniper's anomaly detection effectively prevents unsampled objects from being falsely reported as a leak. More importantly, the path-biased sampling is intrusive and memory consuming, thus it cannot be used in production environment. It does not make sense to spend much more memory to detect leaks in production environment.

Page Protection Based Sampling: Novark et al. present Hound that removes the heavyweight instrumentation for the *staleness* updates using a page-level sampling [106]. The basic idea is to employ a memory protection mechanism of an OS kernel to detect the accesses of the objects. Hound periodically protects every page and updates the last access time of all objects on the same page to the *protection* time. Once a page fault occurs, Hound catches the signal and unprotects it for a performance reason; here, Hound does not update the last access time of all objects on that page until it gets protected again. That is, actual staleness updates are always delayed to the *protection* time. The resulting staleness is underestimated thus posing a risk of false negatives.

Another cause of false negatives is that Hound works at the granularity of a page; it is possible that a page contains both live and dead objects, and a single access to a live object can cause a reset to the staleness of *dead* objects in that page. To mitigate that, Hound changes the underlying memory allocator to perform an age-based segregation of the objects, which can end up degrading the performance of the memory allocator. Nevertheless, the page-level false sharing can still occur depending on memory allocation patterns.

Sniper versus SWAT/Hound: There are key differences between SWAT/Hound and Sniper. First, Sniper is fully automated whereas others are not. Second, Sniper

does not perturb the original application execution. In contrast, SWAT inserts instrumentation code but also changes the original control flow for the path-biased sampling. Hound changes the original memory allocator, which is not acceptable in production runs due to the resulting allocation/deallocation speed and heap size increase. Besides, some applications are tightly coupled with the original memory allocator, thus they may simply fail to run with a new allocator.

Third, Sniper does not require any recompilation while SWAT relies on binary translation. Fourth, Sniper is more robust against false positives/negatives due to its application-tailored anomaly detection, and its sampling operates at a much finer granularity compared to Hound's page-level sampling. *In particular, the anomaly detection strategy reduces the possibility that those objects rarely sampled due to Sniper's uniform sampling in a cold path gets falsely reported as a leak*. On the contrary, SWAT/Hound are vulnerable to false positives/negatives due to their *ad hoc*, manual determination of the *staleness* threshold.

Fifth, Sniper is detachable from the application for mission-critical situations. That way all the overheads due to Sniper can be dynamically managed. On the contrary, the code transformed by SWAT permanently resides as a part of the application. Meanwhile, the internal and external fragmentation caused by Hound's memory allocator continuously affects the application performance. Finally, Sniper has much lower time and space overheads compared to SWAT/Hound. Their overheads particularly get worse for multithreaded applications; that is why they deal with only sequential applications. On the contrary, Sniper supports multithreaded applications with very low overhead.

ECC Protection Based Sampling: Qin et al. present a different approach called SafeMem [109]. It first groups heap objects according to their size and the calling contexts of the allocation site, and measures the lifetime of each object. SafeMem relies on the observation that the maximal lifetime of objects in the same group

remains stable and is thus anticipatable. The underlying assumption is that if the lifetime of a certain object is much longer than the expected lifetime of the group it belongs to, then the object is likely to be a leak. To reduce false positives, SafeMem monitors the access to such suspicious objects using an ECC memory protection mechanism; heap data is scrambled and stored in the memory, and the first access to data, which is recognized by the ECC fault, leads to a conclusion that it is not a leak.

However, such a method arrives at a premature conclusion in that the object can end up with a leak even after multiple accesses. To avoid the false negatives, SafeMem keeps watching some objects even their first access by having the ECC fault handler update metadata such as the lifetime and its maximum of the group. Whenever such objects become suspicious, i.e., the lifetime is longer than some threshold, the ECC monitoring is periodically enabled. Thus, SafeMem compromises the memory reliability, which is critical in datacenters.

Besides, SafeMem's excessive memory consumption prevents its use in production environment. There are a couple of reasons for that. First, it maintains various information for each heap object/group. For heap-bound application, such metadata quickly becomes very large. Second, to differentiate a real hardware memory error from the access fault, SafeMem stores the original heap data in a private memory region for a match against the scrambled data. Those end up occupying considerable space, e.g., 57% memory consumption overhead.

7.4 Related Research on Data Structure Acceleration

Over the last several decades, computer architects have investigated various hardware support to accelerate common data structures including trees and graphs [84, 135, 10, 100, 85, 38, 91]. Wu et al. [143] devised ADP (Abstract Datatype Processor),

special hardware acceleration for hash tables and sparse vectors. To improve memory performance of priority queues, Chandra and Sinnen [23] implemented HardwarePQ based on full shift-register architecture [100]. Recently, Bloom et al. [13] proposed a dedicated hardware logic system called HWDS (Hardware Data Structure) as well as an exception model to support large queues.

All these efforts turn out to effectively improve the application performance by accelerating the hot code of the data structure manipulation. However, any commodity processor has not yet supported such special hardware. Even if FPGA-based data structure accelerators might be available, the operating system and compiler support to leverage them must be addressed in the first place for their pervasive adoption. Thus, there is a compelling need to have a software-only approach for accelerating critical data structure operations on commodity processors.

DSO is inspired by the helper threading approach [68, 80, 151, 124, 43, 71, 88, 32, 19]. That is, the helper thread performs the data structure operation on the behalf of the application. Unlike previous works, DSO does not require any hardware supports, and therefore it can be realized on commodity processors to take the cost of performing the expensive operation away from the application. In addition, we presented a compiler algorithm to eliminate redundant synchronization code misplaced by users.

CHAPTER VIII

CONCLUSIONS AND FUTURE RESEARCH

8.1 Conclusions

Data structures are the main focus of program understanding, performance engineering, bug detection, and security enhancement. Thus, it is very important for developers to have a good programming tool so that they can leverage their data structure in a more effective and more robust way. In light of this, this thesis proposed a programming tool suite that includes four new and enhanced components to improve data structure usage. These four components are (1) DDT and MIDAS: data structure detection tools, (2) Brainy: a data structure selection tool, (3) Sniper: a tool to detect memory leaks for data structures, and (4) DSO: a technique to offload expensive data structure operations.

- **Chapter II** implemented a data structure detection tool called DDT. Detecting data structures can help developers to optimize their program. Through dynamic code instrumentation of a program binary, DDT can automatically detect the organization of data in memory as well as the interface functions used to access the data. Once the program execution finishes, the dynamic invariant detection then determines exactly how those functions modify and utilize the data. We demonstrated that DDT is highly accurate across several different implementations of standard data structures.

- **Chapter III** introduced MIDAS, another tool that can detect data structures without relying on the interface functions for a highly optimized program binary. During program execution, MIDAS traces the shape and data invariants of a data structure as DDT does. To keep the invariants against *destructive updates*

when there is no interface boundary, MIDAS automatically filters out those traces generated while a data structure loses its defined shape. To this end, MIDAS can accurately identify data structures and extract their useful properties based on the invariants irrespective of how they are encapsulated, how different their implementations are, and even how optimized the binary is.

- **Chapter IV** presented Brainy, a novel and repeatable methodology for generating machine learning based models to predict what the best data structure implementation is given a program, a set of inputs, and a target architecture. The work introduces a random program generator that is used to train the machine learning models, and demonstrates that these models are more accurate and more effective than previously proposed hand-constructed models based on traditional asymptotic analysis for real-world applications. The experimental results demonstrate that Brainy can achieve significant performance improvement.

- **Chapter V** provided Sniper, an effective memory leak detection tool for C/C++ production software. To the best of our knowledge, Sniper is the first to provide a systematic methodology for accurate leak identification. Sniper automatically determines the *staleness* threshold based on an anomaly detection. As a result, the leak identification is tailored not just for each application but for each allocation site as well. In particular, Sniper is transparent unlike prior tools; it does not change application behaviors by modifying the executable or replacing the original memory allocator. The empirical evaluation demonstrates that Sniper is highly accurate in detecting critical memory leaks in real-world software.

- **Chapter VI** proposed a data structure acceleration technique called DSO. By

offloading a time-consuming data structure operation to a helper thread running on a separate core, users can improve the overall performance of the application with a minimal effort to place necessary synchronization points. DSO achieves effective communication and synchronization by leveraging a concurrent lockfree queue without any hardware support. In particular, our compiler analysis automatically eliminates redundant synchronization code misplaced by users. We realized DSO on commodity processors, and demonstrated that DSO can achieve significant speedups for data structure intensive real-world applications.

8.2 Future Research

This thesis opens up multiple areas of interesting research. This section summarizes the main points and provides the future research directions.

8.2.1 Future Work for Data Structure Detection

The move toward manycore computing is putting increasing pressure on data orchestration in applications. Identifying what data structures are used within an application is a critical path toward application understanding and performance engineering for the underlying manycore architectures.

Based on dynamic invariant detection, DDT has already reported many useful properties that can be used to identify exactly what data structures are being used in an application, as well as to infer what operations are performed on the data structures. We believe that this is the first step in assisting developers to make a better choice for their target architecture and can provide a significant aid for assisting them in parallelizing their applications. With the integration of existing data dependence profiling tools [150, 72], we can extend DDT to detect those data structures that unnecessarily cause data dependence by themselves.

We also plan to extend DDT by integrating cost models, e.g., Brainy's machine

learning based data structure selection models, in order to predict when alternative data structures are better suited for the target application, and providing semi-automatic or speculative techniques to automatically replace poorly chosen data structures.

8.2.2 Future Work for Data Structure Selection

One obvious approach to enhance Brainy is to extend its scope to parallel applications. That is, Brainy can select the best parallel data structures among many candidates. Note that Brainy has already leveraged a repeatable, automated, and systematic training framework to generate machine learning based models for predicting what the best data structure implementation is given a program, a set of inputs, and a target architecture. I.e., building a model that selects the best parallel data structure only requires running the training framework with the new examples that exercise different behaviors of the target parallel data structures.

Another direction to leverage Brainy's model to tune critical properties of a data structure property to improve the overall performance. E.g., our previous work offers diagnostics for the initial size of vectors or hash tables [21, 130]. Especially for large scale enterprise software, it would be interesting to address how to predict the performance of its distributed data structures and to adjust their property affecting the performance. With that in mind, we plan to address the tradeoffs in distributed hash tables, e.g., predicting the speedup a particular application achieves by relaxing the consistency requirements of the distributed hash table.

8.2.3 Future Work for Memory Leak Detection

The success of *staleness* based leak identification depends on the accurate determination of the *staleness*. It is very important to precisely determine the threshold, since it directly impacts the number of false positives and negatives. Sniper has

leveraged its systematic methodology that automatically determines the *staleness* threshold based on an anomaly detection. As a result, the leak identification is performed in an application-specific manner. To improve the accuracy, Sniper currently applies the statistics to each group of objects that are created in the same allocation site. For further improvement on the accuracy, Sniper can leverage high-level data structure information to group objects based on each data structure they belong to and then to separately apply the statistics to each group.

We also believe that our statistical methodology can improve the accuracy of existing *staleness* based leak detection tools. Most of the time, they depend on various sampling techniques to keep the *staleness* tracking overhead low. I.e., apart from the importance of the accurate *staleness* threshold, there is a compelling need to make up the accuracy loss because of the data access sampling. As the number of uncaught accesses due the the sampling increases, the false positive rate also increases in general. Sniper's anomaly based statistical analysis can allow existing tools to reduce the false positive rate even in the presence of over-approximated *staleness* due to their sampling.

8.2.4 Future Work for Data Structure Acceleration

Even if DSO has a large potential to improve the overall performance of the application, an automated approach of generating the necessary code is essential for its pervasive adoption. While DSO provides a complier analysis to eliminate redundant synchronization code misplaced by users, they are currently required to recognize which program points must have a synchronization code to guarantee program correctness. Thus, it is still users that understand what functions possibly access and modify the offloaded data structure while its operation is performed by the helper thread.

In this respect, we plan to address a compiler analysis that can automatically

find such functions and place the synchronization code on their call site. This can be achieved based on interprocedural Mod/Ref analysis [102], e.g., if a function does not modify any data touched in the offloaded data structure operation, there is no need to place synchronization code before the call site of the function. We believe that a scalable and precise points-to analysis is the key for the success of the Mod/Ref analysis. Thus, this future work will involve developing a demand-driven and context-sensitive points-to analysis for C/C++.

REFERENCES

[1] "packet-o-matic." http://www.packet-o-matic.org/.

[2] "Squid: Optimising web delivery." http://www.squid-cache.org/.

[3] "Olden benchmark suite," 2002. http://www.cs.princeton.edu/∼mcc/olden.html.

[4] "Red hat enterprise linux 6," tech. rep., Red Hat, 2012.

[5] ABD-EL-HAFIZ, S. K., "Identifying objects in procedural programs using clustering neural networks," *Automated Software Eng.*, vol. 7, no. 3, pp. 239–261, 2000.

[6] AHO, A., SETHI, R., and ULLMAN, J., *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[7] AREF, M., "Discussions on the LogicBlox Datalog Optimization Engine," 2009. personal communication.

[8] ARREGOCES, M., *Data Center Fundamentals*. Cisco Press, 2003.

[9] BALLARD, L., "Conflict avoidance: Data structures in transactional memory," 2006.

[10] BHAGWAN, R. and LIN, B., "Fast and scalable priority queue architecture for high-speed network switches," in *IN INFOCOM 2000*, pp. 538–547, 2000.

[11] BIENIA, C., KUMAR, S., SINGH, J. P., and LI, K., "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of the 17th PACT*, Oct. 2008.

[12] BJARNE STROUSTRUP AND ALEX STEPANOV, "Standard Container Benchmark," 2009.

[13] BLOOM, G., PARMER, G., NARAHARI, B., and SIMHA, R., "Shared hardware data structures for hard real-time systems.," in *EMSOFT* (JERRAYA, A., CARLONI, L. P., MARANINCHI, F., and REGEHR, J., eds.), pp. 133–142, ACM, 2012.

[14] BOCCHINO, JR., R. L., ADVE, V. S., DIG, D., ADVE, S. V., HEUMANN, S., KOMURAVELLI, R., OVERBEY, J., SIMMONS, P., SUNG, H., and VAKILIAN, M., "A type and effect system for deterministic parallel java," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 97–116, 2009.

[15] BOEHM, H. J., "Space efficient conservative garbage collection," *SIGPLAN Not.*, vol. 39, pp. 490–501, April 2004.

[16] BOND, M. and MCKINLEY, K., "Tolerating memory leaks," in *Proceedings of the 23rd ACM SIGPLAN OOPSLA*, ACM, 2008.

[17] BOND, M. D. and MCKINLEY, K., "Leak pruning," in *Proceeding of the 14th ASPLOS*, pp. 277–288, ACM, 2009.

[18] BOND, M. D. and MCKINLEY, K. S., "Bell: bit-encoding online memory leak detection," in *Proc. of the 12th ASPLOS*, (New York, USA), 2006.

[19] BROWN, J. A., WANG, H., CHRYSOS, G., WANG, P. H., and SHEN, J. P., "Speculative Precomputation on Chip Multiprocessors," in *the 6th Workshop on Multithreaded Execution, Architecture (MTEAC-6)*, November 2002.

[20] BRUENING, D. and ZHAO, Q., "Practical memory checking with dr. memory," in *Proc. of the 9th CGO*, pp. 213–223, 2011.

[21] C. JUNG AND THE GCC TEAM, "GCC, the GNU Compiler Collection, 4.5 Contribution," 2010. http://gcc.gnu.org/news.html.

[22] CARLSTROM, B. D., *Programming With Transactional Memory*. PhD thesis, Stanford University, 2008.

[23] CHANDRA, R. and SINNEN, O., "Improving application performance with hardware data structures," Tech. Rep. 678, Stanford University, Mar. 2010.

[24] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., and GRUBER, R. E., "Bigtable: A distributed storage system for structured data," in *Proc. of 7th USENIX OSDI*, 2006.

[25] CHEN, D., VACHHARAJANI, N., HUNDT, R., LI, X., ERANIAN, S., CHEN, W., and ZHENG, W., "Taming hardware event samples for precise and versatile feedback directed optimizations," *Computers, IEEE Transactions on*.

[26] CHEREM, S., PRINCEHOUSE, L., and RUGINA, R., "Practical memory leak detection using guarded value-flow analysis," in *Proc. of 28th PLDI'07*.

[27] CHILIMBI, T. M., "Heapmd: Identifying heap-based bugs using anomaly detection," in *In International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[28] CHILIMBI, T. M. and HAUSWIRTH, M., "Low-overhead memory leak detection using adaptive statistical profiling," in *Proc. of 11th ASPLOS'04*.

[29] CHILIMBI, T. M., HILL, M. D., and LARUS, J. R., "Cache-conscious structure layout," pp. 1–12, May 1999.

[30] CHUNG, I.-H., *Towards Automatic Performance Tuning*. PhD thesis, University of Maryland, College Park, 2004.

[31] CLAUSE, J. and ORSO, A., "Leakpoint: pinpointing the causes of memory leaks," in *Proc. of the 32nd ICSE*, (New York, NY, USA), 2010.

[32] COLLINS, J. D., TULLSEN, D. M., WANG, H., and SHEN, J. P., "Dynamic speculative precomputation," in *Proceedings of The 34th International Symposium on Microarchitecture (MICRO-34)*, December 2001.

[33] COZZIE, A., STRATTON, F., XUE, H., and KING, S., "Digging for Data Structures," pp. 255–266, 2008.

[34] CVE DETAILS, "Common Vulnerabilities and Exposures (CVE)," 2013. http://www.cvedetails.com.

[35] CWE DETAILS, "Common Weakness Enumeration (CWE)," 2013. http://cwe.mitre.org/data/definitions/401.html.

[36] DEAN, J. and GHEMAWAT, S., "Mapreduce: simplified data processing on large clusters," in *Proc. of 5th USENIX OSDI*, 2004.

[37] DEKKER, R. and VERVERS, F., "Abstract data structure recognition," in *Knowledge-Based Software Engineering Conference*, pp. 133–140, Sept. 1994.

[38] DELORIMIER, M., KAPRE, N., MEHTA, N., RIZZO, D., ESLICK, I., RUBIN, R., URIBE, T. E., KNIGHT, T. F., and DEHON, A., "Graphstep: A system architecture for sparse-graph algorithms," in *In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. IEEE*, IEEE Computer Society, 2006.

[39] DEMSKY, B. and RINARD, M. C., "Goal-directed reasoning for specification-based data structure repair," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 931–951, 2006.

[40] DEMSKY, B., ERNST, M. D., GUO, P. J., MANT, S. M., PERKINS, J. H., and RINARD, M. C., "Inference and enforcement of data structure consistency specification s," in *International Symposium on Software Testing and Analysis*, pp. 233–244, 2006.

[41] DIMACS, "The 9th dimacs implementation challenge - shgortest paths." http://www.dis.uniroma1.it/ challenge9/.

[42] DING, S. Q. and XIANG, C., "Overfitting problem: a new perspective from the geometrical interpretation of mlp," pp. 50–57, 2003.

[43] DING, Y., KANDEMIR, M., RAGHAVAN, P., and IRWIN, M., "A helper thread based edp reduction scheme for adapting application execution in cmps," pp. 1–14, 2008.

[44] DONGARRA, J., LONDON, K., MOORE, S., MUCCI, P., and TERPSTRA, D., "Using papi for hardware performance monitoring on linux systems," in *Proceedings of the 2nd International Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*, 2001.

[45] DRONGOWSKI, P. J., "Instruction-based sampling: A new performance analysis technique for amd family 10h processors," 2007.

[46] DUBACH, C., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., and TEMAM, O., "Fast compiler optimisation evaluation using code-feature based performance prediction," in *ACM International Conference on Computing Frontiers*, 2007.

[47] EDWARD WUSTENHOFF, "Service Level Aggrement in the Data Center," 2002.

[48] ERANIAN, S., *Perfmon2: a Standard Performance Monitoring Interface*.

[49] ERNST, M. and OTHERS, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, pp. 35–45, Dec. 2007.

[50] GANTZ, J., CHUTE, C., MANFREDIZ, A., MINTON, S., REINSEL, D., SCHLICHTING, W., and TONCHEVA, A., "The diverse and exploding digital universe," 2008. International Data Corporation.

[51] GHEMAWAT, S., GOBIOFF, H., and LEUNG, S.-T., "The google file system," in *Proc. of the 19th SOSP*, (New York, NY, USA), 2003.

[52] GHIYA, R. and HENDREN, L. J., "Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 1996.

[53] GOOGLE, "Google code search," 2009. http://www.google.com/codesearh.

[54] GUSFIELD, D., *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[55] HASSOUN, M. H., *Fundamentals of Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1995.

[56] HASTINGS, R. and JOYCE, B., "Purify: Fast detection of memory leaks and access errors,"

[57] HEINE, D. L. and LAM, M. S., "A practical flow- and context-sensitive c/c++ memory leak detector," in *Proc. of the 23rd PLDI*, 2003.

[58] HIND, M., "Pointer analysis: haven't we solved this problem yet?," pp. 54–61, June 2001.

[59] HUSSEIN, F., "Genetic algorithms for feature selection and weighting, a review and study," in *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*, (Washington, DC, USA), p. 1240, 2001.

[60] Intel Corporation, CA, *Intel®Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide*, 2010.

[61] ITRS, "International technology roadmap for semiconductors exectutive summary, 2008 update," 2008. http://www.itrs.net/Links/2008ITRS/Update/2008_Update.pdf.

[62] JARMULAK, J. and CRAW, S., "S.: Genetic algorithms for feature selection and weighting. in," in *Proceedings of the IJCAI'99 workshop on Automating the Construction of Case Based Reasoners*, pp. 28–33, 1999.

[63] JOURNAL OF INSTRUCTION LEVEL PARALLELISM, "3rd workshop on computer architecture competitions: Memory scheduling championship," 2012.

[64] JULA, A. and RAUCHWERGER, L., "Two memory allocators that use hints to improve locality," in *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, (New York, NY, USA), pp. 109–118, ACM, 2009.

[65] JUMP, M. and MCKINLEY, K. S., "Dynamic shape analysis via degree metrics," in *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, (New York, NY, USA), pp. 119–128, ACM, 2009.

[66] JUNG, C., LIM, D., LEE, J., and HAN, S., "Adaptive execution techniques for SMT multiprocessor architectures," pp. 236–246, 2005.

[67] JUNG, C. and CLARK, N., "Ddt: design and evaluation of a dynamic program analysis for optimizing data structure usage," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 56–66, ACM, 2009.

[68] JUNG, C., LIM, D., LEE, J., and SOLIHIN, Y., "Helper thread prefetching for loosely-coupled multiprocessor systems," in *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, (Washington, DC, USA), pp. 140–140, IEEE Computer Society, 2006.

[69] JUNG, C., WOO, D.-K., KIM, K., and LIM, S.-S., "Performance characterization of prelinking and preloading for embedded systems," in *Proc. of the 7th ACM & IEEE EMSOFT*, (New York, NY, USA), 2007.

[70] JUNG, Y. and YI, K., "Practical memory leak detector based on parameterized procedural summaries," in *Proc. of the 7th ISMM*, 2008.

[71] KIM, D. and YEUNG, D., "Design and Evaluation of Compiler Algorithms for Pre-Execution," in *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pp. 159–170, October 2002.

[72] KIM, M., KIM, H., and LUK, C.-K., "Sd3: A scalable approach to dynamic data-dependence profiling," in *43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010, 4-8 December 2010, Atlanta, Georgia, USA*, pp. 535–546, IEEE, 2010.

[73] KOTSIANTIS, S. B., "Supervised machine learning: A review of classification techniques.," *Informatica (Slovenia)*, vol. 31, no. 3, pp. 249–268, 2007.

[74] KULKARNI, M. and OTHERS, "Optimistic Parallelism Requires Abstractions," pp. 211 – 222, June 2007.

[75] KUNCAK, V., LAM, P., ZEE, K., and RINARD, M. C., "Modular pluggable analyses for data structure consistency," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 988–1005, 2006.

[76] LAMPORT, L., "Specifying concurrent program modules," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 190–222, Apr. 1983.

[77] LATTNER, C. and ADVE, V., "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," pp. 75–86, 2004.

[78] LATTNER, C. A., *Macroscopic data structure analysis and optimization*. PhD thesis, Champaign, IL, USA, 2005. Adviser-Adve, Vikram.

[79] LEATHER, H., BONILLA, E., and O'BOYLE, M., "Automatic Feature Generation for Machine Learning Based Optimizing Compilation," Mar. 2009.

[80] LEE, J., JUNG, C., LIM, D., and SOLIHIN, Y., "Prefetching with helper threads for loosely coupled multiprocessor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 9, pp. 1309–1324, 2009.

[81] LEE, J., PARK, J.-H., KIM, H., JUNG, C., LIM, D., and HAN, S., "Adaptive execution techniques of parallel programs for multiprocessors," *J. Parallel Distrib. Comput.*, vol. 70, pp. 467–480, May 2010.

[82] LEE, J., WU, H., RAVICHANDRAN, M., and CLARK, N., "Thread Tailor : Dynamically Weaving Threads Together for Efficient , Adaptive Parallel Applications," *Language*, 2010.

[83] LEE, S. and TUCK, J., "Parallelizing Mudflap using Thread-Level Speculation on a Chip Multiprocessor," pp. 72–80, 2008.

[84] LEE, S., TIWARI, D., SOLIHIN, Y., and TUCK, J., "Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor.," in *HPCA*, pp. 99–110, IEEE Computer Society, 2011.

[85] LEISERSON, C., SCIENCE., C.-M. U. P. P. D. O. C., and DEPT, C.-M. U. C. S., *Systolic Priority Queues*. Defense Technical Information Center, 1979.

[86] LIAO, C., QUINLAN, D. J., WILLCOCK, J. J., and PANAS, T., "Extending automatic parallelization to optimize high-level abstractions for multicore," in *IWOMP '09: Proceedings of the 5th International Workshop on OpenMP*, pp. 28–41, 2009.

[87] LIU, L. and RUS, S., "perflint: A Context Sensitive Performance Advisor for C++ Programs," Mar. 2009.

[88] LUK, C.-K., "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," in *Proceedings of the 29th International Symposium on Computer Architecture*, June 2001.

[89] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., and HAZELWOOD, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of the 25th PLDI*.

[90] MALHOTRA, V. and KOZYRAKIS, C., "Library-based prefetching for pointer-intensive applications," 2006.

[91] MENCER, O., HUANG, Z., and HUELSBERGEN, L., "Hagar: Efficient multi-context graph processors," in *Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, FPL '02, (London, UK, UK), pp. 915–924, Springer-Verlag, 2002.

[92] MENS, T. and TOURWE, T., "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[93] MEREDITH, B., "Omega: An instant leak detector tool for Valgrind," 2008. http://www.brainmurders.eclipse.co.uk/omega.html.

[94] MICHAEL PROCOPIO, "Cloud computing does not require virtualization," 2011. http://www.enterprisecioforum.com.

[95] MICROSOFT SHAREPOINT FOUNDATION, "Hyper-V Performance Tests," 2010. http://technet.microsoft.com/en-us/library/gg454734.aspx.

[96] MITCHELL, N., SCHONBERG, E., and SEVITSKY, G., "Four trends leading to java runtime bloat," *IEEE Software*, vol. 27, pp. 56–63, 2010.

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

[98] MITCHELL, T. M., *Machine Learning*. New York: McGraw-Hill, 1997.

[99] MONSIFROT, A., BODIN, F., and QUINIOU, R., "A machine learning approach to automatic production of compiler heuristics," in *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pp. 41–50, 2002.

[100] MOON, S.-W., SHIN, K. G., and REXFORD, J., "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Trans. Comput.*, vol. 49, pp. 1215–1227, Nov. 2000.

[101] MOZILLA.ORG, "Mozilla Bugzilla," 2012. https://bugzilla.mozilla.org.

[102] MUCHNICK, S., *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[103] NAGAPPAN, N., MAXIMILIEN, E. M., BHAT, T., and WILLIAMS, L., "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Softw. Eng.*, 2008.

[104] NETHERCOTE, N. and SEWARD, J., "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proc. of the 28th PLDI*, 2007.

[105] NIKOLAEV, R. and BACK, G., "Perfctr-xen: a framework for performance counter virtualization," in *Proc. of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, 2011.

[106] NOVARK, G., BERGER, E. D., and ZORN, B. G., "Efficiently and precisely locating memory leaks and bloat," in *Proc. of the 30th PLDI*, 2009.

[107] PAYNE, B. D., *Improving Host-Based Computer Security Using Secure Active Monitoring and Memory Analysis*. PhD thesis, Georgia Institute of Technology, 2010.

[108] PUSUKURI, K. K., GUPTA, R., and BHUYAN, L. N., "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, (Washington, DC, USA), pp. 116–125, IEEE Computer Society, 2011.

[109] QIN, F., LU, S., and ZHOU, Y., "Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs," in *Proc. of the 11th HPCA*, 2005.

[110] QUILICI, A., "Reverse engineering of legacy systems: a path toward success," in *Proceedings of the 17th International Conference on Software Engineering*, pp. 333–336, 1995.

[111] RAMAN, E. and AUGUST, D. I., "Recursive data structure profiling," in *ACM SIGPLAN Workshop on Memory Systems Performance*, June 2005.

[112] REN, G., TUNE, E., MOSELEY, T., SHI, Y., RUS, S., and HUNDT, R., "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4.

[113] RUGINA, R., "Quantitative shape analysis," in *In Static Analysis Symposium (SAS04*, pp. 228–245, SpringerVerlag, 2004.

[114] RUMELHART, D. E., HINTON, G. E., and WILLIAMS, R. J., "Learning internal representations by error propagation," pp. 673–695, 1988.

[115] S. ERANIAN, "Perfmon2: a flexible performance monitoring interface for linux," in *In Ottawa Linux Symposium (OLS)*, 2006.

[116] SAGIV, M., REPS, T., and WILHELM, R., "Parametric Shape Analysis via 3-Valued Logic," *ACM Trans. Programming Languages and Systems*, vol. 24, no. 3, pp. 217–298, 2002.

[117] SAGIV, M., REPS, T., and WILHELM, R., "Solving shape-analysis problems in languages with destructive updating," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, (New York, NY, USA), pp. 16–31, ACM, 1996.

[118] SCHONBERG, E., SCHWARTZ, J. T., and SHARIR, M., "An automatic technique for selection of data representations in setl programs," *ACM Trans. Program. Lang. Syst.*, vol. 3, pp. 126–143, April 1981.

[119] SCHWARTZ, J. T., "Automatic data structure choice in a language of very high level," in *POPL '75: Proceedings of the 2nd ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 36–40, 1975.

[120] SHACHAM, O., VECHEV, M., and YAHAV, E., "Chameleon: adaptive selection of collections," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pp. 408–418, 2009.

[121] SIEDLECKI, W. and SKLANSKY, J., "A note on genetic algorithms for large-scale feature selection," *Pattern Recogn. Lett.*, vol. 10, no. 5, pp. 335–347, 1989.

[122] SLEATOR, D. D. and TARJAN, R. E., "Self-adjusting binary search trees," *J. ACM*, vol. 32, no. 3, pp. 652–686, 1985.

[123] SNEVELY, R., *Enterprise Data Center Design and Methodology*. Prentice Hall, 2002.

[124] SOLIHIN, Y., LEE, J., and TORRELLAS, J., "Using a user-level memory thread for correlation prefetching," in *Proceedings of the 29th International Symposium on Computer Architecture*, May 2002.

[125] STEPANOV, A. and LEE, M., "The standard template library," tech. rep., WG21/N0482, ISO Programming Language C++ Project, 1994.

[126] STEPHENSON, M. W., *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, 2006.

[127] STLPORT STANDARD LIBRARY PROJECT, "Standard C++ Library Implementation for Borland C++ Builder 6 (STLport)," 2009.

[128] TANG, Y., TANG, Y., GAO, Q., GAO, Q., QIN, F., and QIN, F., "Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages," in *Proc. of USENIX 2008 Annual Technical Conference*.

[129] THE APACHE SOFTWARE FOUNDATION, "The Apache C++ Standard Library (STDCXX)," 2009.

[130] THE GCC TEAM, "GCC C++ standard library," 2010. http://gcc.gnu.org/libstdc++.

[131] THE GCC TEAM, "GCC, the GNU Compiler Collection," 2010. http://gcc.gnu.org.

[132] THE GNOME PROJECT, "GLib 2.20.0 Reference Manual," 2009.

[133] TRIANTAFYLLIS, S., BRIDGES, M. J., RAMAN, E., OTTONI, G., and AUGUST, D. I., "A framework for unrestricted whole-program optimization," in *In ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pp. 61–71, 2006.

[134] TRIMARAN, "An infrastructure for research in ILP," 2000. http://www.trimaran.org/.

[135] TSAY, J.-J., "An efficient implemention of priority queues using fixed-sized systolic coprocessors," *Inf. Process. Lett.*, vol. 46, no. 4, pp. 193–198, 1993.

[136] UPADHYAYA, G., MIDKIFF, S. P., and PAI, V. S., "Using data structure knowledge for efficient lock generation and strong atomicity," in *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 281–292, ACM, 2010.

[137] WANG, Z. and O'BOYLE, M. F., "Mapping parallelism to multi-cores: a machine learning based approach," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 75–84, 2009.

[138] WHITTAKER, J., *How to Break Software Security*. Addision Wesley.

[139] WILCOX, R., *Introduction to Robust Estimation and Hypothesis Testing*. Elsevier Science & Technology, 2012.

[140] WILHELM, R., SAGIV, M., and REPS, T., "Shape analaysis," Mar. 2000.

[141] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., and DEMMEL, J., "Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms," 2007.

[142] WIRTH, N., *Algorithms + Data Structures = Programs*. Prentice Hall, 1978.

[143] WU, L., KIM, M., and EDWARDS, S., "Cache impacts of datatype acceleration," *IEEE Comput. Archit. Lett.*, vol. 11, pp. 21–24, Jan. 2012.

[144] XIE, Y. and AIKEN, A., "Context- and path-sensitive memory leak detection," in *In Proc. of ESEC/FSE 2005*, ACM Press, 2005.

[145] XU, G., BOND, M. D., QIN, F., and ROUNTEV, A., "Leakchaser: helping programmers narrow down causes of memory leaks," in *Proc. of the 32nd PLDI*, 2011.

[146] XU, G., MITCHELL, N., ARNOLD, M., ROUNTEV, A., and SEVITSKY, G., "Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (New York, NY, USA), pp. 421–426, ACM, 2010.

[147] XU, G. and ROUNTEV, A., "Precise memory leak detection for java software using container profiling," in *Proc. of the 30th ICSE*, 2008.

[148] XU, G. and ROUNTEV, A., "Detecting inefficiently-used containers to avoid bloat," in *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, ACM, 2010.

[149] ZEE, K., KUNCAK, V., and RINARD, M., "Full functional verification of linked data structures," pp. 349–361, June 2008.

[150] ZHANG, X., NAVABI, A., and JAGANNATHAN, S., "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, (Washington, DC, USA), pp. 47–58, IEEE Computer Society, 2009.

[151] ZILLES, C. B. and SOHI, G. S., "Execution-based prediction using speculative slices," in *Proceedings of The 28th International Symposium on Computer Architecture (ISCA'01)*, July 2001.