Toward Automatic Data Structure Replacement for Effective Parallelization

Changhee Jung and Nathan Clark
College of Computing
Georgia Institute of Technology
{cjung9, ntclark}@cc.gatech.edu

ABSTRACT

Data structures define how values being computed are stored and accessed within programs. By recognizing what data structures are being used in an application, tools can make applications more robust by enforcing data structure consistency properties, and developers can better understand and more easily modify applications to suit the target architecture for a particular application.

This paper presents the design and application of DDT, a new program analysis tool that automatically identifies data structures within an application. A binary application is instrumented to dynamically monitor how the data is stored and organized for a set of sample inputs. The instrumentation detects which functions interact with the stored data, and creates a signature for these functions using dynamic invariant detection. The invariants of these functions are then matched against a library of known data structures, providing a probable identification. That is, DDT uses program consistency properties to identify what data structures an application employs. The empirical evaluation shows that this technique is highly accurate across several different implementations of standard data structures.

1. INTRODUCTION

Data orchestration is rapidly becoming the most critical aspect of developing effective manycore 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 [11]. 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 tenfold every five years [8]. 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 [14]. 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 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?

And 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 [15]. *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 the Daikon invariance detection tool [7] 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 splay-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. The information about data structure usage can be fed into performance modeling tools, which can estimate when alternative data structures may be better suited for an application/architecture, or simply used by developers to better understand the performance characteristics of the applications they are working on.

We have implemented DDT as part of the LLVM toolset [13] and tested it on several real-world data structure libraries: the GNOME C Library (GLib) [23], the Apache C++ Standard Library (STDCXX) [22], Borland C++ Builder's Standard Library implementation (STLport) [21], and a set of data structures used in the Trimaran research compiler [25]. This work demonstrates that DDT is quite accurate in detecting data structures no matter what the implementation.

2. RELATED WORK

There is a long history of work on detecting how data is organized in programs. Shape analysis (e.g., [9, 19, 26]) 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 [12, 27], 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 [6]. These properties can then be used to automatically detect bugs, repair data structures online, or improve many other software engineering tasks [5]. 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 add resilience to 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.*

Similar to this goal, work by Guo et al. tries to automatically derive higher-level information from a program [16]. In Guo's work, a program analyzer uses dataflow analysis to monitor how variables interact. The interactions are used to synthesize subtypes in situations where a single primitive type, such as integer, could have multiple meanings that are not meant to interact, such as distance and temperature. Our work is different in that we are not inferring types, so much as we are recognize the functionality provided by complex groups of variables. Identifying data structures involves not only separating values into partitioned groups, but also identifying interface functions to the structure and recognizing what the interface functions do. Both Guo's technique and our technique have important uses and can be leveraged to improve programmer understanding of the application.

The reverse-engineering community has also done work similar to this effort [1, 17]. 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 three works most similar to ours are by Raman et al. [18], Dekker et al. [4], and Cozzie et al. [3]. 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 constrained, (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 ex-

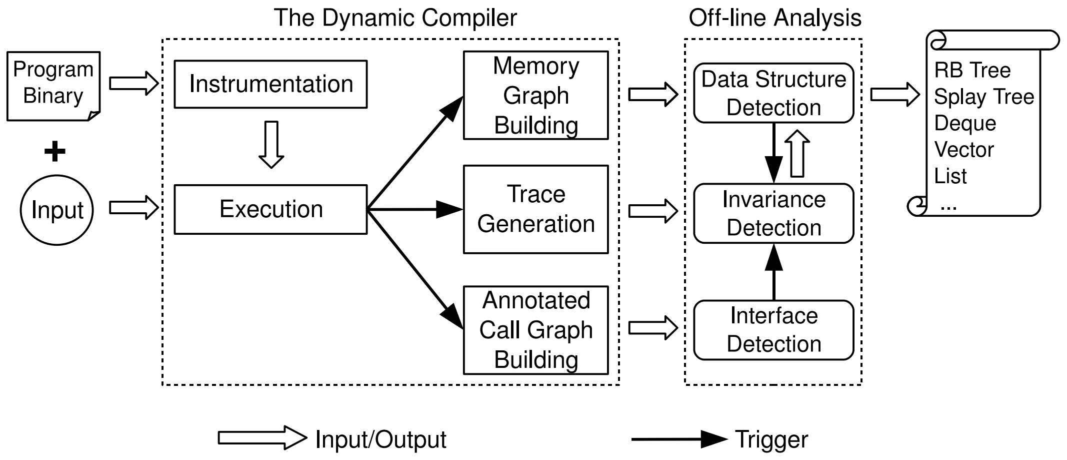


Figure 1: Structure of DDT.

ecution 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 sets implemented as 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.

The following are the contributions of this paper:

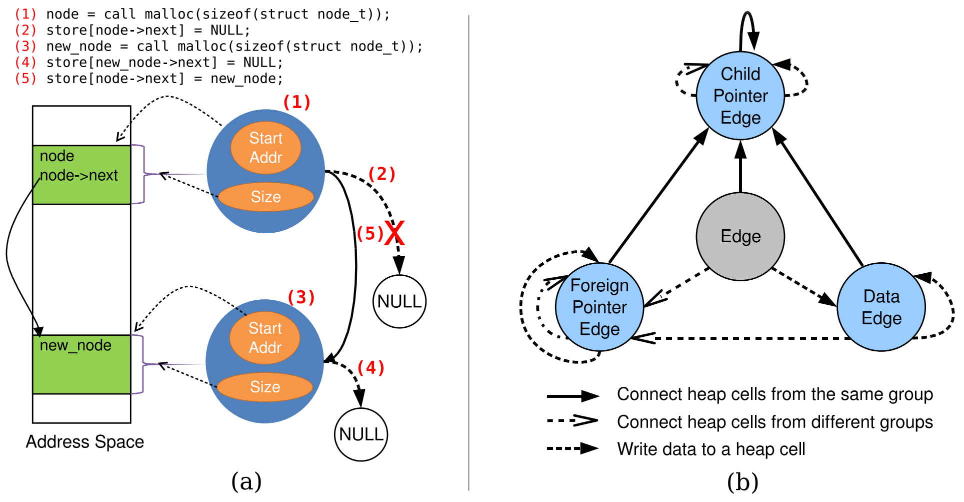
- *A new approach to identifying data structures:* DDT dynamically monitors the memory layout of an application, and detects interface functions that access or modify the layout. Invariant properties of the memory graph and interface functions are matched against a library of known data structures, providing a probabilistic identification. This approach significantly improves on previous work, as it is less conservative, does not require source code access, and is not dependent on data structure implementation.
- *An empirical evaluation demonstrating DDT's effectiveness:* We test the effectiveness of DDT on several real-world data structure libraries and show that, while unsound, this analysis is both reasonably fast and highly accurate. DDT can be used to help programmers understand performance maladies in real programs, which ultimately helps them work with the compiler and architecture to choose the most effective data structures for their systems.

3. DATA STRUCTURE IDENTIFICATION 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 the annotated call graph. (3) Understanding what those functions do; invariants on the memory organization and interface functions are the basis for characterizing how the data structure operates.

Figure 1 shows a high-level diagram of DDT. An application binary and sample inputs 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 decision tree within the library 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.



```
(1) node = call malloc(sizeof(struct node_t));
(2) store[node->next] = NULL;
(3) new_node = call malloc(sizeof(struct node_t));
(4) store[new_node->next] = NULL;
(5) store[node->next] = new_node;
```

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.

3.1 Tracking Data Organization with a Memory Graph

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 to each region occur. 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 binary trees 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 presented in previous work [18]. 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 [10]. 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.

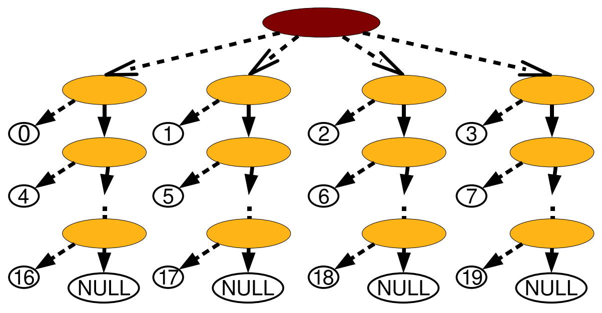
However, in the many-to-one mapping, an allocation site typically belongs to one data structure, but one data structure might have many allocation sites. In order to correctly identify the data structure in such a situation, it is necessary to merge the memory node types. This merging can be done by 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

[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. 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.

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 when 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.

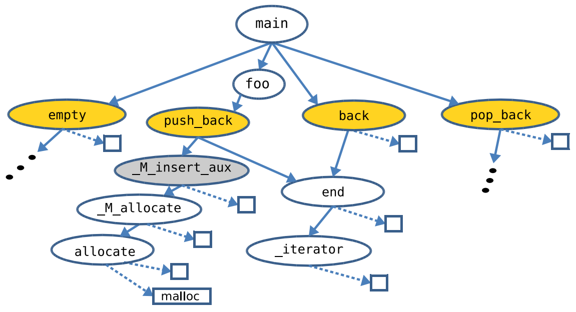
Figure 3 gives an example demonstrating why typing nodes and edges in the memory graph is critical in recognizing data structures. The code snippet in this figure creates a `vector` with four `lists` and inserts integer numbers between 0 and 19 into each `list` in a round robin manner. Nodes are colored 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 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 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.

3.2 Identifying Interface Functions for the Memory Graph

Understanding how data is organized through the memory graph is the first step toward automatically 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 mem-



(a)

```
void foo (int num)         template<class T>
{                          void vector<T>::
    // num is mutable          push_back(const T& x)
    num = ...              {
                               // x is immutable


    ...                        ...



    v.push_back(num);          _m_insert_aux(end(), x);




}                          }
```

(b)

Figure 4: (a) Interface extraction from a dynamic call graph for the STL `vector` class and (b) code snippet showing argument immutability.

ory 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 [20]. 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 the function and determines if they are modified during the function using either dataflow analysis or invariant detection. If there are no immutable 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 [24]. 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. 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.

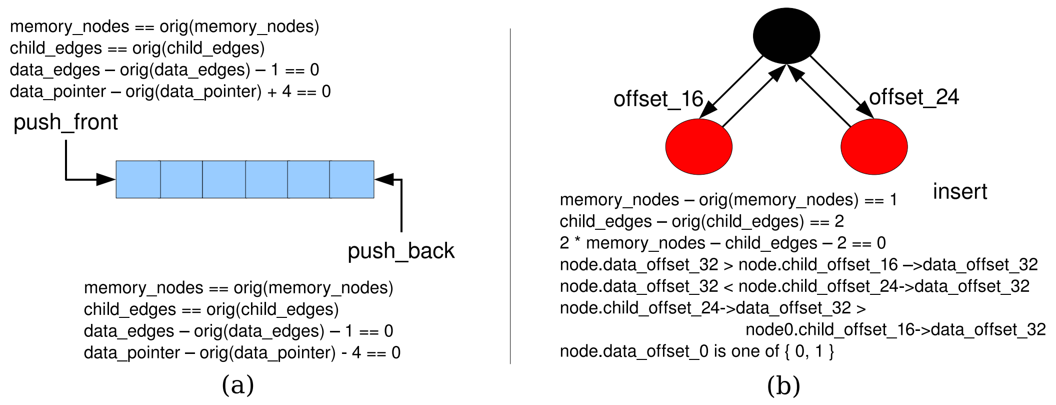

memory_nodes == orig(memory_nodes)
child_edges == orig(child_edges)
data_edges – orig(data_edges) – 1 == 0
data_pointer – orig(data_pointer) + 4 == 0
push_front
push_back
memory_nodes == orig(memory_nodes)
child_edges == orig(child_edges)
data_edges – orig(data_edges) – 1 == 0
data_pointer – orig(data_pointer) - 4 == 0
(a)
offset_16
offset_24
insert
memory_nodes – orig(memory_nodes) == 1
child_edges – orig(child_edges) == 2
2 * memory_nodes – child_edges – 2 == 0
node.data_offset_32 > node.child_offset_16 –>data_offset_32
node.data_offset_32 < node.child_offset_24->data_offset_32
node.child_offset_24->data_offset_32 >
node0.child_offset_16->data_offset_32
node.data_offset_0 is one of { 0, 1 }
(b)

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

3.3 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 [7].

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 3.1, 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 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 [7] 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.

[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.

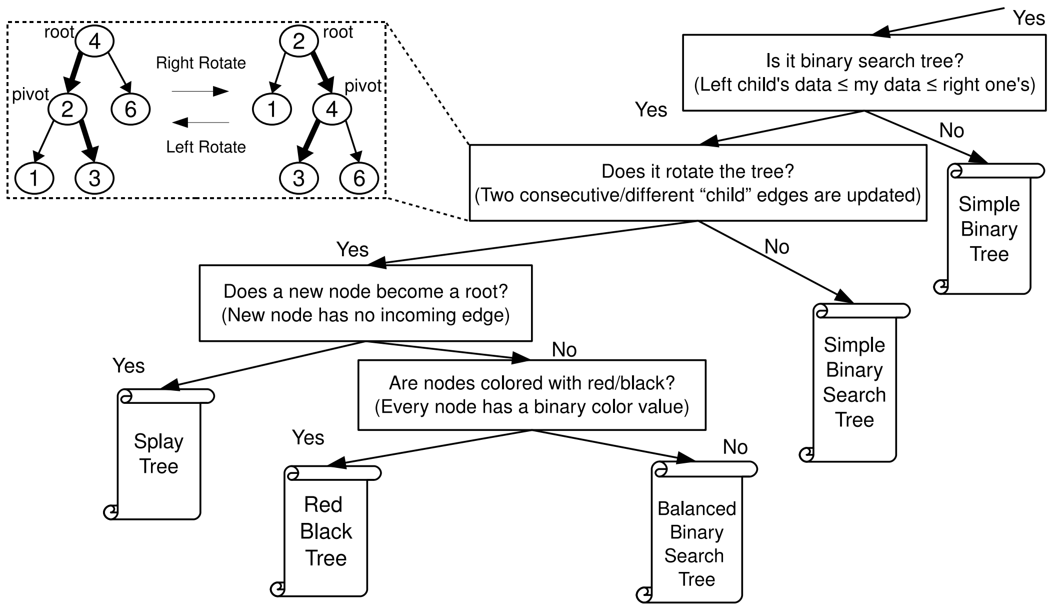

root
4
Right Rotate
2
root
pivot
2
6
Left Rotate
1
4
pivot
1
3
3
6
Yes
Is it binary search tree?
(Left child's data ≤ my data ≤ right one's)
Yes
No
Does it rotate the tree?
(Two consecutive/different "child" edges are updated)
Simple Binary Tree
Yes
No
Does a new node become a root?
(New node has no incoming edge)
Yes
No
Simple Binary Search Tree
Are nodes colored with red/black?
(Every node has a binary color value)
Splay Tree
Yes
No
Red Black Tree
Balanced Binary Search Tree

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

3.4 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.

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 3.3. 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 empirically demonstrates DDT can effectively detect different implementations from several real-world data structure libraries.

4. EVALUATION

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

In order to demonstrate the utility of DDT, we implemented it as part of the LLVM toolset. DDT 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 [7] 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) [20]. 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 [23], the Apache C++ Standard Library STDCXX [22], Borland C++ Builder's Standard Library STLport [21], and a set of data structures used in the Trimaran research compiler [25].

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.

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 [2], 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.

5. 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.

6. REFERENCES

[1] S. K. Abd-El-Hafiz. Identifying objects in procedural programs using clustering neural networks. *Automated Software Eng.*, 7(3):239–261, 2000.

[2] Bjarne Stroustrup and Alex Stepanov. Standard Container Benchmark, 2009.

[3] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for Data Structures. In *Proc. of the 2008 on Operating Systems Design and Implementation*, pages 255–266, 2008.

[4] R. Dekker and F. Ververs. Abstract data structure recognition. In *Knowledge-Based Software Engineering Conference*, pages 133–140, Sept. 1994.

[5] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis*, pages 233–244, 2006.

[6] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering*, 32(12):931–951, 2006.

[7] M. Ernst et al. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, Dec. 2007.

[8] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva. The diverse and exploding digital universe, 2008. International Data Corporation.

[9] R. Ghiya and L. J. Hendren. 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.

[10] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of the 2001 ACM Workshop on Program Analysis For Software Tools and Engineering*, pages 54–61, June 2001.

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

[12] V. Kuncak, P. Lam, K. Zee, and M. C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006.

[13] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[14] S. Lee and J. Tuck. Parallelizing Mudflap using Thread-Level Speculation on a Chip Multiprocessor. In *Proc. of the 2008 Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, pages 72–80, 2008.

[15] L. Liu and S. Rus. perflint: A Context Sensitive Performance Advisor for C++ Programs. In *Proc. of the 2009 International Symposium on Code Generation and Optimization*, Mar. 2009.

[16] Philip J. Guo and Jeff H. Perkins and Stephen McCamant and Michael D. Ernst. Dynamic Inference of Abstract Types. pages 255–265, 2006.

[17] A. Quilici. Reverse engineering of legacy systems: a path toward success. In *Proceedings of the 17th International Conference on Software Engineering*, pages 333–336, 1995.

[18] E. Raman and D. August. Recursive data structure profiling. In *Third Annual ACM SIGPLAN Workshop on Memory Systems Performance*, June 2005.

[19] M. Sagiv, T. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[20] A. Stepanov and M. Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1994.

[21] STLport Standard Library Project. Standard C++ Library Implementation for Borland C++ Builder 6 (STLport), 2009.

[22] The Apache Software Foundation. The Apache C++ Standard Library (STDCXX), 2009.

[23] The GNOME Project. GLib 2.20.0 Reference Manual, 2009.

[24] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *In ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 61–71, 2006.

[25] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[26] R. Wilhelm, M. Sagiv, and T. Reps. Shape analaysis. In *Proc. of the 9th International Conference on Compiler Construction*, Mar. 2000.

[27] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *Proc. of the SIGPLAN '08 Conference on Programming Language Design and Implementation*, pages 349–361, June 2008.