Recursive Data Structure Profiling

Easwaran Raman David I. August

Department of Computer Science
Princeton University
Princeton, NJ 08544
{eraman,august}@princeton.edu

ABSTRACT

As the processor-memory performance gap increases, so does the need for aggressive data structure optimizations to reduce memory access latencies. Such optimizations require a better understanding of the memory behavior of programs. We propose a profiling technique called *Recursive Data Structure Profiling* to help better understand the memory access behavior of programs that use recursive data structures (RDS) such as lists, trees, etc. An RDS profile captures the runtime behavior of the individual instances of recursive data structures. RDS profiling differs from other memory profiling techniques in its ability to aggregate information pertaining to an entire data structure instance, rather than merely capturing the behavior of individual loads and stores, thereby giving a more global view of a program's memory accesses.

This paper describes a method for collecting RDS profile without requiring any high-level program representation or type information. RDS profiling achieves this with manageable space and time overhead on a mixture of pointer intensive benchmarks from the SPEC, Olden and other benchmark suites. To illustrate the potential of the RDS profile in providing a better understanding of memory accesses, we introduce a metric to quantify the notion of *stability* of an RDS instance. A stable RDS instance is one that undergoes very few changes to its structure between its initial creation and final destruction, making it an attractive candidate to certain data structure optimizations.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Measurement techniques]

General Terms

Experimentation, Measurement

Keywords

RDS, dynamic shape graph, list linearization, memory profiling, shape profiling

1. INTRODUCTION

The continuing trend of deeper processor pipelines and the increasing gap between memory speed and the processor speed necessitates new techniques for memory latency tolerance. To develop these techniques, a high-level understanding of the memory characteristics of programs is required. That is, we need to understand how programmer intended to use the memory, not just how the individual load/store operations in the program behave. Static analysis techniques like alias analysis and shape analysis help us understand how a program uses memory. Unfortunately these techniques are conservative and are not intended to capture the dynamic memory behavior of applications, which is necessary for developing more aggressive optimizations. Dynamic memory behavior of programs is recorded by memory profilers, but existing memory profilers typically operate at the granularity of individual memory operations or memory addresses. As a result, they do not provide the kind of high-level understanding of memory behavior desirable for any potential aggressive memory optimizations of the future.

To help guide new memory optimizations, we want to develop a profiling technique that overcomes the above mentioned drawbacks of existing memory profiling schemes. Since address-regular memory accesses, like array traversals, are usually better understood and easier to optimize than irregular accesses, we focus our efforts on the latter. In particular, our focus is on the dynamic memory characteristics of recursive data structures (RDS). RDSs are created by data types that are defined in terms of themselves. The ideas described in this paper are not dependent on any particular programming language, but for the ease of understanding, we use examples from the C programming language. In C, RDS are a special case of what is known as Linked Data Structures(LDS). A Linked Data Structure is created by a C structure that has a pointer field. By making this pointer point to an object of the same structure type, RDSs are formed.

Consider some examples that illustrate our terminology. An array of pointers to integers creates an LDS, since these pointers serve as links. This, however, is not considered as an RDS. On the other hand, a list node structure that has a pointer field to the same list node structure would produce an RDS. An RDS can also be mutually recursive when a structure of type A has a pointer to structure of type B and vice-versa. Continuing with our list example, a program can create many separate lists from the same list structure by having many 'head' pointers pointing to the start of the lists. We use the term *RDS instance* to denote these separate lists with separate head pointers. We use the term *RDS type* to denote the set of data structure declarations that create all these separate list instances.

Since the size of RDSs is unbounded due to their recursive nature, RDSs can form a major part of the irregular memory access in a program. Hence, we propose a technique called Recursive Data Structure Profiling to study the dynamic memory behavior of these structures without requiring any high level representation or type information thereby enabling its application even on legacy applications. This constitutes the main contribution of this paper.

As a demonstration of the RDS profiler's ability to provide new ways to understand the memory access behavior, we introduce the notion of *RDS stability* and a metric to quantify it. Informally, a stable RDS is one which, once created, suffers "few" changes to its structure during its lifetime. We quantify this informal notion by defining a metric called the RDS stability factor. This notion of stability is crucial in the development of optimizations, like *list linearization* [2, 10], that attempt to remap the data structure to a different location in memory during runtime. If an RDS is stable, then this remapping has to be done only once after its creation, and the benefits of this remapping will not be lost due to changes to the RDS instance.

This paper is organized as follows. In the next section, we describe related work. Section 3 describes intuitively how the RDS instances are identified without using type information, and presents the RDS profiling algorithm in detail. In Section 4, we give detailed information on our profiler framework implementation. Section 5 describes some of the properties of RDS that are captured by the profiler. Then, in Section 6, we tabulate these properties for a set of benchmarks. We also report the space and time overhead of our profiler in this section. Finally, in Section 7 we conclude and list the future work.

2. RELATED WORK

There have been various works on memory analysis, memory profiling, and profile based optimizations, but most of them work at the granularity of individual memory operations. To our knowledge, there exists no prior work on memory profiling at the RDS instance granularity. In this section, we first examine related works in memory analysis, then in profiling techniques closer to our work, and finally prior work establishing a need for RDS profiling.

Lattner and Adve [7, 8] provide a link-time analysis on their LLVM framework to identify logically disjoint data structures in a program. Their analysis produces a *disjoint data structure graph*, which is a graph representation of all the data structures in the program. Since this is a static analysis, the solution is a conservative one. Our approach is a profile-based one that identifies only those data structure instances that appear in a particular run of the program. Moreover, the crucial difference is that their analysis requires the type information provided by LLVM, precluding it from being applied on executables, whereas our profiler does not require any type information.

Shape analysis [14, 4, 5] is a compile-time analysis technique that characterizes the *shape* of the data structures, that is, properties like sharing of nodes, cyclicity and reachability. Radu [13] proposed a method called quantitative shape analysis. This method computes quantitative properties like the skew and height for trees in cases where they are computable at compile time, thereby conveying more information about the shapes than prior methods. All these compile-time analysis schemes are conservative, cannot operate on multiple compilation units and, in most cases, the analysis time does not scale well with the size of the program. Moreover, as already mentioned, they do not provide information on the runtime characteristics of these shapes.

Wu et al. [16] proposed a scheme called *object-relative memory profiling*, where an object corresponds to the memory allocated by a single call to a memory allocation routine. The objects are assigned unique identifiers that are then used in the profile results instead of memory addresses. In [16], objects allocated by separate calls to the memory allocator but linked to each other by pointers are not grouped together. In contrast, we treat heap locations of same type connected by pointers as one logical entity and generate a profile at that granularity. Calder et al.[1] perform memory profiling to enable cache-conscious data placement. They construct two profiles called *name* and *temporal relationship graph*. The former is related to the idea of object-relative memory profiling. The temporal relationship graph captures the temporal relationship between accesses to heap locations. Again, this does not give the programmer-intended view of the heap locations while our technique does.

Nystrom et al. [11] have characterized the access patterns of recursive data structures in integer benchmarks. They use a metric called *data access affinity* to study the correlation among accesses to pointer-chasing loads. This only gives a local view of the shape graph. Moreover their scheme depends on compiler annotations to track the links and their traversal, while we use some instrumentation and then track the flow at runtime.

While RDS profiling is likely to open up many possibilities for new optimizations, at least one existing optimization would benefit. This optimization, known as list linearization, was first proposed in the context of the LISP programming language [2]. Luk and Mowry [10], describe this optimization in the context of C programs. They suggest applying this optimization one or many times depending on whether the list under consideration is altered or not. The RDS stability metric we propose provides a way of identifying this property, thereby allowing an automatic way of applying this optimization.

3. COLLECTING RDS PROFILES

In this section, we describe the methodology of collecting the RDS profiles, without going into the specifics of our implementation. These details will be given in Section 4. The steps involved in collecting an RDS profile are:

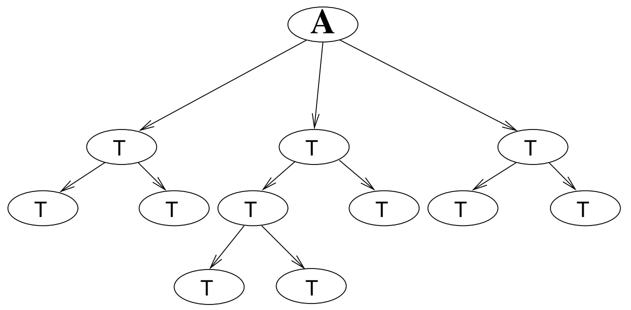
- Reconstructing the *shape graphs* (defined below)
- Associating events with shape graphs

3.1 Terminology

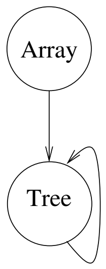
We now define some terminology which we use in the rest of the paper. An *alloc call* is a call to any procedure that allocates memory from the heap region. This can be any such procedure from the standard C library – `malloc, calloc or realloc` – or any other user defined procedure that allocates heap memory. An *object id (OID)* is an identifier that uniquely identifies a chunk of memory obtained by a single call to an alloc routine. The OID consists of two components: a static id, which uniquely identifies the alloc call site, and a dynamic id(*dynid*) that uniquely identifies every *instance* of an alloc call. We can construct a graph from an RDS instance by treating memory chunks allocated by individual alloc calls as nodes and the pointers to other nodes as edges. We call such graphs *Shape Graph*s.

3.2 Reconstruction of the shape graphs

An important step in RDS profiling is the on-the-fly reconstruction of shape graphs created during program execution by observing the execution. Before describing how shape graphs are reconstructed, let us first define them precisely. A shape graph is a *connected* directed graph $G = (V, E)$. The set of vertices V is a set of dynamically allocated objects of the *same RDS type*. An edge $\langle u, v\rangle \in E$ if and only if a pointer field in u points to v. Since a



a. Unified Shape Graph



b. Static Shape Graph

Figure 1: Array of Trees. USG and the corresponding SSG

particular RDS type declaration can have multiple instances created at run time, it can produce multiple shape graphs.

Identification of heap objects We first assign a unique identifier to every heap-allocated memory location. The identifier should not only be unique but also contain information about the location of the static instruction of the alloc call, for reasons that will be explained later. Identifiers are generated by inserting instrumentation at each alloc call site. This only requires the binary executable and the symbol table.

Identification of the links between the heap objects Once the heap objects are identified, we need to identify how the heap objects are linked together. An edge is created whenever there is a store instruction of the form

```
store r1[off]= r2
```

where the registers r1 and r2 contain addresses from the heap area. Thus, to identify the links, we need to track the flow of heap-generated addresses as the program executes. There are at least two ways of tracking this by instrumenting the binary appropriately, as we will see in Section 4.

We can construct a graph whose adjacency list representation is specified by the list of links identified as above. Such a graph might contain nodes from different RDS instances. To further explain this, let us consider an array of trees. Figure 1(a) shows a directed graph corresponding to an array of trees with both the tree nodes and the array node. The nodes labeled T are the tree nodes and the node labeled A is the array node that was created dynamically. Instead of treating this whole graph as a single entity, we want to separate the different instances of the tree, each of which is a subgraph of the graph in Figure 1(a). To achieve this, we develop an algorithm using the properties of these graphs based on two simple observations. Before stating those observations, let us define two more graphs: a *unified shape graph* (USG) and a *static shape graph* (SSG). An USG is the graph that is described above, whose adjacency list representation contains the set of all links in the program. Formally, a USG is a graph $G = (V, E)$, where V is the set of dynamically allocated heap objects and E is the set of pointer links between the elements of the set V. An SSG is a graph $G' = (V', E')$, where V' is the set of all static alloc call-sites in the program and an edge $e' = (u', v') \in E'$ if $e = (u, v) \in E$ and u and v are heap locations allocated by the call sites u' and v' respectively. The USG and the SSG corresponding to the array of trees example is given in Figure 1.

Any dynamically allocated linked data structure created in the program can be represented by an induced subgraph in the SSG. The alloc calls that create the data structure form the nodes of this induced subgraph. Our first observation is that such a subgraph corresponding to a *recursive* data structure of unbounded size forms a strongly connected component (SCC) in the SSG.[1] This is because, while an RDS instance could have potentially unbounded nodes that are connected to each other, the SSG has only a finite number of nodes. This creates a cycle in the graph, leading to a SCC. This situation is similar to representing recursion in a call graph: potentially unbounded invocations of a set of calls are represented by a small set of call-graph nodes leading to an SCC in the call graph. Based on this observation, an RDS type corresponds to an SCC in the static shape graph. We note that there are ways of creating an RDS that produce an induced subgraph which is not a single SCC. Consider the case where two different lists are created by two different list creation routines and connected together. The resulting induced subgraph is not a SCC, but it contains two SCCs. We treat these as two different RDS types.

The second observation is related to the individual instances of an RDS type. Two different RDS instances of an RDS type are always separated by nodes that do not belong to that type: if they are not, then they are, by definition, the same instance. In other words, if only the nodes of a particular RDS type are retained in the USG, the different RDS instances of that type will form disjoint connected components, as any connection between them would be *only* through nodes of a different type. For example, if we retain only the tree type RDS nodes in 1, the different instances of the tree type would not be connected to each other and all nodes in the same instance would form a connected component (ignoring the edge orientations).

These two properties of the RDS lead to an algorithm for identifying individual instances. A naïve algorithm would be to collect the entire USG to a trace file and later process the graph to identify the RDS instances based on these properties. This approach soon becomes infeasible when collecting certain properties of the RDS instances. For example, consider the lifetime (the time between the creation of the first node and the deletion of the last node) of an RDS instance. To compute this information using the naïve algorithm, one must keep track of the lifetime information of *all* the edges in the USG and later summarize it during the post-processing phase. Thus, even though the useful data – lifetime in this case – is just 4 or 8 bytes per RDS instance, we would be collecting that much data *per* edge in the naïve algorithm. This would result in a huge increase in the size of the trace when a program contains a few RDS instances with a large number of nodes.

As we will see later, this problem can be avoided if we are able to categorize the edges of the USG into the RDS instances to which they belong, on the fly. We need to keep track of those connected components of the USG that correspond to the RDS instances. Identifying connected components can be efficiently implemented using union-find data structure [3]. We treat two nodes of the USG as connected only if they have an edge between them and the corre-

[1]For the purpose of our algorithm, we do not consider a single node without a self-loop as an SCC.

```
typedef struct _TREE {
    int n; struct _TREE * left, *right;
} tree;


tree *make_tree (int depth){
    if(depth >0){
        tree *t = (tree *)malloc(sizeof(tree));
        t->n = depth;
        t->left = make_tree(depth-1);
        t->right= make_tree(depth-1);
    }
    else{
        return NULL;
    }
}
int main(int argc, char **argv){
    tree **arr = (tree **)malloc(10*sizeof(tree *));
    arr[0] = make_tree(2);
    arr[1] = make_tree(2);
    return 0;
}
```

(a) Source code

```
make_tree:
        ...
1       cmp4.ge p6, p7 = 0, r32
2       (p6) br.cond.dptk .L32
3       br.call.sptk.many b0 = malloc
                                ;static id : T

4       mov r33 = r8
        ...
5       br.call.sptk.many b0 = make_tree
6       adds r14 = 8, r33
7       st8 [r14] = r8
        ...
8       adds r33 = 16, r33
9       br.call.sptk.many b0 = make_tree
10      st8 [r33] = r8
.L32:
        ...
11      br.ret.sptk.many b0
main:
        ...
12      br.call.sptk.many b0 = malloc

                                ;static id : A

13      mov r32 = r8
        ...
14      br.call.sptk.many b0 = make_tree
15      st8 [r32] = r8, 8
        ...
16      br.call.sptk.many b0 = make_tree
17      st8 [r32] = r8
        ...
18      br.ret.sptk.many b0
```

(b) Relevant portions of the IA-64 assembly code

Figure 2: A program that creates two trees and stores the pointers to the root in a dynamically created array

sponding nodes in the SSG belong to the same connected component. For example, in Figure 1, even though the node labeled A and a node labeled T have an edge between them, we don't place them in the same connected component as the corresponding SSG nodes do not belong to the same SCC. So we also maintain the SCC information in the static shape graph along with the union-find data structure. When a new USG edge is seen, the nodes of the edge are mapped to the node(s) in the SSG by making use of the static id component of the OID, and a corresponding edge is created in the SSG if it does not exist already. Then, we check if those node(s) in the SSG belong to the same SCC, in which case we use the union-find data structure to do a join of the two nodes. On the other hand, if the static nodes do not belong to the same SCC, then all we know is that at this point in the program's execution, we cannot conclude that they are in the same SCC. But a later edge might make them belong to the same SCC and so we have to remember these edges without summarizing them. If a change occurs in the SCC of the SSG, then these remembered edges are revisited to see if they have to be merged.

This process is illustrated in Figure 3. The C code and the relevant portions of the assembly code for that example are given in Figure 2. The `main` function allocates an array of tree pointers dynamically, creates balanced trees of depth 2, and assigns the resulting tree pointers to the the first two elements of the array. In Figure 3, the left column shows the dynamic instruction trace of this program, with only the instructions relevant to the tree creation shown. The next column shows the assignment of unique dynamic ids (dynid) to the result of alloc calls. In this calling convention, the register `r8` contains the return value of the function calls. We show the dynid corresponding to the registers that contain the heap addresses. The next section shows how we implement this in our profiler. The third column shows the formation of the USG and the next column shows how SSG evolves. The edges in the USG are created when both the address and the value of a store instruction are heap addresses. The action taken on edge creation is shown in the fifth column, and the resulting set of RDS instances are shown in the final column. On encountering the edge $1 \rightarrow 2$, we connect their corresponding SSG nodes, which is the same node T in this case. Since this forms a SCC (trivially), we know that the nodes 1 and 2 are of the same RDS type. We merge the profile information from these nodes and keep track of the fact that the elements 1 and 2 belong to the same instance. This is shown in the last column, where a set $\{1,2\}$ is created and is treated as a separate RDS instance. Similarly, when the edge $1 \rightarrow 3$ is seen, 1 and 3 are merged together, and the set $\{1,2\}$ is augmented to contain the element 3. When the edge $0 \rightarrow 1$ is seen, we notice that the corresponding static graph nodes of 0 and 1 (A and T) are in different SCCs. Therefore, we do not merge these two nodes but instead put that edge in a queue so that later, if T and A become part of the same SCC, we can merge the nodes 0 and 1. When the next edge, $4 \rightarrow 5$ is created, a new shape graph instance is created to contain 4 and 5, since the corresponding static node T forms an SCC. Note that these two nodes (4 and 5) are not merged with the existing set $\{1, 2, 3\}$, as there is no edge connecting elements from these two sets. Similarly, the set $\{4, 5\}$ is augmented to include 6 after the next store operation. The final store creates an edge between 0 and 4, but since the corresponding static nodes A and T are still in different SCCs, they are not merged. At the end of the example, we are left with two sets of RDS instances- $\{1, 2, 3\}$ and $\{4, 5, 6\}$. These correspond to the two instances of the tree in the program, which are the only two RDS instances in the program.

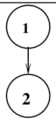
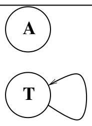
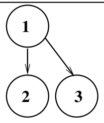
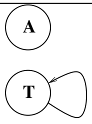
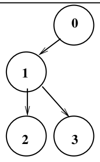
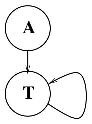
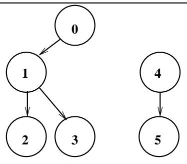
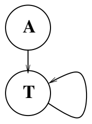
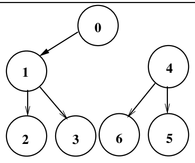
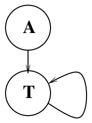
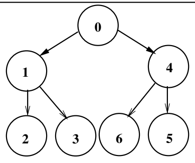
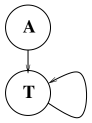
Instruction trace	dynid	USG	SSG	Action	RDS instances
12: br.call malloc 13: mov r32 = r8 . . . 3: br.call malloc 4: mov r33 = r8 . . . 3: br.call malloc . . . 6: adds r14 = 8, r33	dynid[r8] = 0 dynid[r32] = 0 dynid[r8] = 1 dynid[r33] = 1 dynid[r8] = 2 dynid[r14] = 1				
7: st8 [r14] = r8				merge(1,2) since both map to static node T	{1,2}
. . . 8: adds r33 = 16, r33 . . . 3: br.call malloc . . .	 dynid[r33] = 1 dynid[r8] = 3				
10: st8[r33] = r8 . . .				merge(1,3) since both map to static node T	{1,2,3}
15: st8 [r32] = r8,8				add the edge (0,1) to a queue since T and A are not in SCC yet	
. . . 3: br.call malloc 4: mov r33 = r8 . . . 3: br.call malloc . . . 6: adds r14 = 8, r33	 dynid[r8] = 4 dynid[r33] = 4 dynid[r8] = 5 dynid[r14] = 4				
7: st8 [r14] = r8				merge(4,5)	{1,2,3},{4,5}
. . . 8: adds r33 = 16, r33 . . . 3: br.call malloc . . .	 dynid[r33] = 4 dynid[r8] = 6				
10: st8[r33] = r8 . . .				merge(4,6)	{1,2,3},{4,5,6}
17: st8 [r32] = r8				add edge (0,4) to a queue since T and A are not in same SCC yet	

Figure 3: Example illustrating the working of our algorithm for an array of trees

3.3 Associating events with shape graphs

Once the RDS instances are identified, any metric of an event of interest during program execution could be profiled at the granularity of RDS instance if we could establish a mapping between the event and an RDS instance. Let us consider the example of cache misses during traversals of an RDS instance. The events of interest are the execution of load operations whose address and data are both heap memory locations. Since such a load traverses an edge in the USG, it gets mapped to the RDS instance that contains this edge, if any. The metric we are interested in is a boolean value indicating if the event results in a cache hit or a miss. Since multiple loads might be mapped to a single RDS instance, we also need a function to aggregate this event in a suitable way. In this example, the function is just a **sum** function that adds the cache misses due to different loads together. These aggregation functions are used to combine the contents of the auxiliary data structure during the join operation in the union-find data structure.

4. IMPLEMENTATION

We now describe our framework (Figure 4) to collect the RDS profile. The profiler is built using Pin [9], an instrumentation framework for IA-64 binaries.

To track the nodes of the USG, we instrument the program by inserting `nop` instructions that have special meaning to the emulator. These `nop` instructions convey information about the type of the alloc call (`malloc`, `realloc` etc.) and the static id of that alloc call to the emulator. When the alloc call executes, the emulator associates an OID with the address generated by the alloc. If the contents of a storage element (register or memory location) has an OID associated with it, it implies that the storage element contains an address in the heap region. This OID information is used during the execution of stores to determine if the stores create the edges of the USG and during the execution of loads to determine if it is a pointer-chasing load. To obtain the OIDs corresponding to the operands of the loads and stores, two approaches could be followed. One is to let the OIDs flow along the datapath, as illustrated in the example in the previous section. This could be implemented by maintaining a shadow register file with OIDs and keeping track of heap addresses stored in memory. The other approach is to maintain a mapping between the heap locations and the OIDs in a suitable data structure and query the structure during load and store instructions to obtain their OIDs. The second approach is much simpler to implement than the first one, though it has a minor drawback: the contents of a storage element might not have been obtained by an alloc call, but still resemble a heap address whose OID information is stored. For example, this could happen when a large immediate value loaded in a register lies in the range of heap addresses. But this has a low probability of occurrence, especially in architectures with 64 bit addressing, and so we choose the second approach and use a balanced binary tree to map the addresses to the object ids. For the applications we have chosen, we have verified that no spurious edge is introduced in the SSG by this method.

Our profiler framework (Figure 4) consists of two components:

- the OID manager
- the profile builder

The OID manager and the profile builder closely interact with each other to produce the RDS profile. We now describe these two components, their functionalities, and the interactions between them.

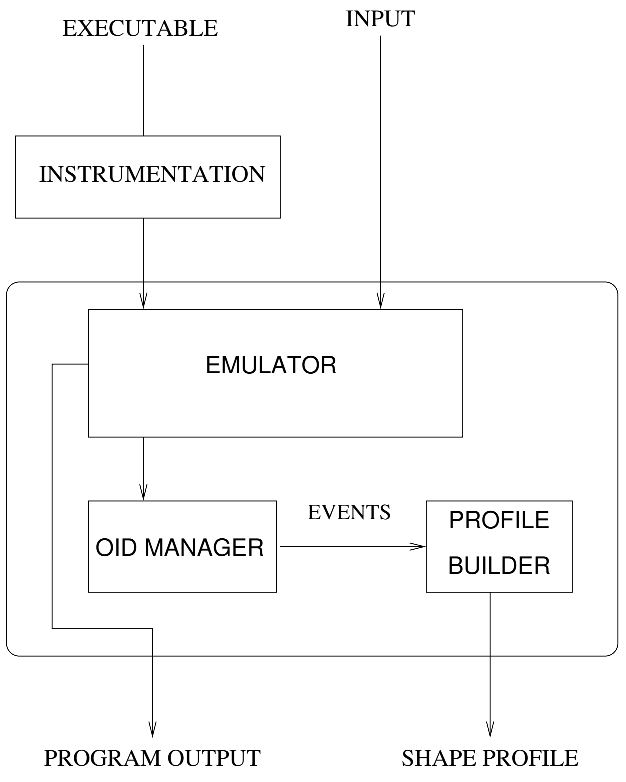


Figure 4: Block diagram of the profiler

4.1 OID manager

The function of the OID manager is to manage the OIDs generated by the alloc call. The OIDs are generated by instrumenting the system calls that allocate memory from the heap: `malloc`, `calloc`, and `realloc` or any other user-defined alloc call. The immediate field of the `nop` instruction provides the static id part of the OID, while the dynid is generated by a counter incremented after every alloc call. On every `malloc` and `calloc` (or the equivalent call) the current value of the counter is used to form the OID and then the counter is incremented. Since a `realloc` merely alters the size of an existing object and does not create a new "logical" object, it reuses the counter value from the OID corresponding to its input heap address. The mapping between the heap locations and the OIDs are maintained in an AVL tree. Each node of this tree contains the heap address generated by some alloc call, the number of bytes allocated by that call, its OID and its dynamic instruction count. We use the dynamic instruction count as a representative of the execution time.

On a store instruction, the OID manager obtains the OIDs corresponding to both the store address and the store value from the AVL tree. If both the address and the value have a valid OID, it generates the *edge_add* event in the profile builder that indicates that a USG edge has been created. The source and destination OIDs and the offset of the source node at which the link originates are passed to the profile builder along with this event. On a load instruction, if the load address and the loaded value have a valid OID, the OID manager generates the *edge_traverse* event passing the same values as in the case of *edge_add*. On a call to the `free` routine, which is also appropriately instrumented, the OID manager generates the *node_delete* event passing the OID of the deleted node. Thus the OID manager maintains the OID information, determines if a USG edge is created or traversed or if a USG node is deleted, and triggers appropriate events in the profile builder.

4.2 Profile builder

The profile builder receives the edges of the USG from the OID manager and uses them to reconstruct shape graphs and collect the profile. The OID manager triggers the *edge_add*, *edge_traverse* and the *node_delete* events, signifying addition of edges, traversal of edges and removal of nodes on loads, stores, and calls to `free` respectively. These events are implemented as procedure calls in the profile builder.

The profile builder maintains and updates the static shape graph. It also maintains the connected component information using the union-find data structure. The basic union-find data structure is modified so that each node is also associated with a pointer to an auxiliary data structure that is used for the purposes of profile collection, as described in 3.3

On an *edge_add* event, the profile builder obtains the static id information of the two nodes from the OIDs and creates an edge between the nodes with these static ids in the SSG, if an edge does not exist already. Note that the static nodes corresponding to the two dynamic nodes can be the same, in which case the resulting edge creates a self loop in the SSG. Then the profile builder checks to see if the nodes belong to the same SCC in the static shape graph. Identifying strongly connected components in a graph can be done in $O(|V| + |E|)$ time [3]. Typically, the SSG is of a small size and so the cost of identifying SCCs by this method will not be high. But we can do better than this since the graph changes only incrementally, one edge at a time. We use the online algorithm for finding SCCs given by Pearce and Kelly ([12]). By maintaining certain information, the algorithm ensures that only a section of the graph has to be searched for the presence of a new SCC when a new edge is added. This algorithm has a complexity $O(\delta \log \delta)$, where δ is proportional to the size of the section of the graph that has to be searched when this edge is inserted. After updating this SCC information, we check if the two static nodes belong to the same SCC. If so, we merge these two nodes using the union-find data structure. We also merge the the auxiliary information of the two nodes appropriately.

On an *edge_traverse* event, the representative node corresponding to the two nodes is found from the union-find data structure. The metrics of interest associated with this event are suitably combined with the contents that already exist in the auxiliary data structure.

On the *node_free* event, the profile builder updates the fact that a particular node has been removed. This is used in computing the RDS lifetime information. This event could also be used to reduce the space requirement by using the union find with delete [6] structure.

5. SCOPE OF RDS PROFILING

In this section we discuss a subset of metrics of RDS instances that can be collected using RDS profiling. These metrics reveal useful information about RDS and their memory access pattern that are not revealed by existing profiling techniques.

Lifetime of an RDS instance. The lifetime of an RDS instance is the time between its creation and destruction. There are many ways of defining the creation and destruction of an RDS instance. We consider the time when the first node in the RDS is allocated as the creation time and the time when the RDS instance is last traversed as the destruction time. The lifetime of an RDS instance is an important criterion in estimating the cost/benefit trade-offs involved in applying any dynamic optimizations at RDS granularity.

Edge properties. We can collect various metrics involving the RDS edges. For example we classify the edges as forward or backward edges depending on whether the source of the edge is older than the destination or vice-versa. This property provides an understanding of how the RDSs are created. An RDS instance with lots of backward edges is created bottom up. This information could be used while designing cache prefetchers for linked data structures. For example a stride based prefetcher might use negative strides while traversing RDS instances created bottom-up.

Operations involved in RDS creation. When the oid manager triggers events to the profile builder, it can also pass information on the static instruction in the program that triggered the event. This helps to collect all the instructions involved in the creation, traversal and deallocation of the nodes and edges in an RDS instance.

Shape of the RDS. In our experiments, rather than maintaining the shape graph in its entirety, we only store the information about the connected components. If we retain the RDS instance as a graph, we could identify the actual shape by some post-processing. But some of the edges in this graph may be transient. For example, a list reversal routine might produce cycles in an RDS instance even though the list may not have cycles otherwise. One heuristic to alleviate this problem is to add an edge to the shape graph only if it is not replaced by another edge that originates from the same node at the same offset within a particular interval. Choosing the interval appropriately will remove the transient edges.

Traversal patterns. Another interesting application of shape profiling is to identify the traversal patterns of RDSs. For a given RDS instance, we try to find correlation between successive traversals of that instance. As an example, if an access $u \rightarrow v$ is followed by $u' \rightarrow v'$, we can categorize this sequence based on whether $v = u'$ or $u = u'$ or no relationship exists between the vertices. This helps determine whether a DFS or a BFS is the more likely traversal of the graph.

Memory performance of RDS instances. RDS profiling captures the memory performance of RDS instances. Data layout optimizations can use this information to layout only those RDS that incurs significant memory access latencies. The performance of different memory allocators can also be compared based on this metric.

RDS stability factor. An important property of an RDS is a measure of their *stability*. The notion of stability is an useful metric for doing list linearization [2, 10]. For linearization to give maximum benefits, the pointer fields of the list must not change after the list is linearized.

A stable structure is one where the relative positions of the RDS elements is unchanged once the edges are created for the first time. Thus, stability measures how *array like* an RDS is as the relative positions of the elements are never changed in an array. As an example, a linked list in which an element is never inserted is considered stable.

To quantify this notion of stability, we propose a new metric called *stability factor*. In order to compute this metric, we first divide the lifetime of the instance by marking n alteration points along its lifetime, where an alteration point is a program point where a new edge is added to the RDS instance or an edge is removed from the instance. We denote the number of accesses between the points i and $i+1$ as $a(i)$. The RDS Stability Factor (RSF) s is defined as

$$s = \min(k | (\sum_{j \in i_1, i_2 \dots i_k} a(j)) \geq t.A)$$

where A is the total number of pointer chases in that instance and t is some threshold close to 1. In our experiments, we set t to be 0.99. An RDS with a stability factor of 1 indicates that atleast 99% of all its pointer chasing loads take place in an interval where

L1D	16K, 4 way associative, 1 cycle latency
L2 Unified	256K, 8 way associative, 6 cycle latency
L3	1.5M, 12 way associative, 13 cycle latency
Memory	100 cycle latency

Table 1: Details of the cache hierarchy

there are no stores to the pointer field of any of the RDS nodes in that instance. An RDS with a lower RDF is a better candidate for applying linearization.

6. EXPERIMENTAL RESULTS

The profiler is implemented using Pin [9] for IA-64 binaries. The experiments were conducted on a 900MHz Itanium 2 machine with 2GB RAM running RH7.1 Linux. For the experiments that involve measuring the memory access latency, we use a cache simulator developed using the Liberty Simulation Environment (LSE) [15]. The simulator models a four-level functional hierarchy and emulates IA-64 binaries. The details of the memory hierarchy are shown in Table 1.

We ran the RDS profiler on a mix of SPEC2000, Olden and two other benchmarks – *ks*, an implementation of a graph partitioning algorithm, and *tree_puzzle*, which implements a fast tree search algorithm – that use recursive data structures. The dynamic instructions executed by the applications are given in Table 2. We first show the performance of the profiler in terms of its space and time overhead. Then we show some characteristics of the benchmarks themselves that are revealed by RDS profiling.

6.1 Profiler performance

For each benchmark, time taken to emulate the benchmark with and without the RDS profiler. The values are given in columns 2 and 3 of Table 2.

The memory requirements for the profiler consist of three major components. The first component is the space required to store the AVL tree that tracks the OID. The number of nodes is bounded by the maximum number of allocs at any point in time. The second component is the size of the union-find data structure. The number of entries in this structure is also bounded by the maximum number of allocs. The third component is the size of the structures for storing the profile information for individual RDS instances. Unlike the other two components, the size of this is proportional only to the number of the shape graphs, which is usually a much smaller value than the number of allocs.

The memory requirement is given in the fourth column of Table 2. We note that most of the benchmarks have a very low space requirement (<1MB). In contrast, tree_puzzle takes up to 153 MB of memory. The memory requirement depends on the RDS usage of the applications.

6.2 Memory characteristics of applications

We now discuss the memory characteristics of the different applications we have used in this experimental setup. The properties of the RDS that we measure are tabulated in Table 3. The benchmarks in our suite show a wide range of RDS properties. This wide range of behavior among pointer intensive routines illustrate the need for further understanding their behavior by techniques like RDS profiling.

The first property we quantify is the *type* of RDS. As discussed earlier, the type of the RDS corresponds to a strongly connected component in the static shape graph. There are a small number of RDS types in many of the programs. Most of them have just one or two RDS types, with *197.parser* having a maximum of 31 RDS types. But each type might have multiple instances created at runtime. The number of RDS instances show a large variation between the benchmarks. Among the SPEC benchmarks, on one side of the spectrum *197.parser* creates more than a million RDS instances, while *130.li* has just one RDS instance. In the next two columns we partition the edges in the shape graphs into forward and backward edges as defined in the previous section. Such a categorization indicates whether the data structures are created in a top-down fashion or a bottom-up fashion. The next column shows the average size of an RDS instance measured in number of edges. The average size in number of edges of an RDS instance also shows a lot of variance ranging from 5 in *175.vpr* to more than 3 million in *130.li*. The table also shows the total accesses of the edges of the shape graph and the average latency to traverse an edge for the given cache model. As expected, long-running benchmarks with a few long-lived shapes have low average access latency per RDS instance, due to high locality.

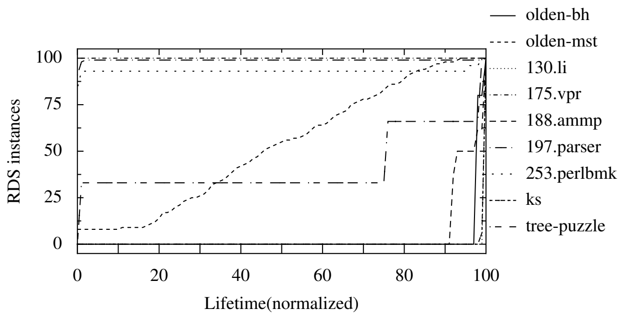


Figure 5: Cumulative distribution of RDS lifetimes

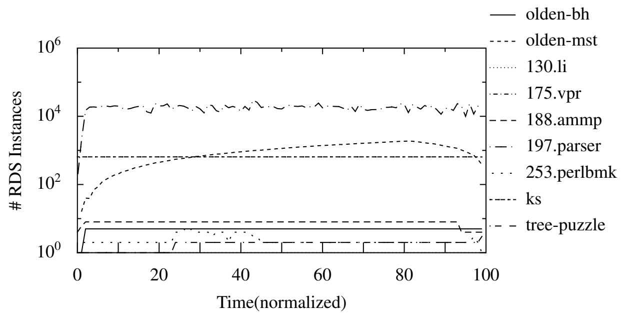


Figure 6: Time vs # RDS instances

6.2.1 *Distribution of RDS lifetime*

We now take a detailed look at the lifetime of RDS instances. Figure 5 shows the cumulative distribution frequency of the lifetimes. The X axis shows the time normalized with respect to the total execution time of the program and the Y axis shows the cumulative distribution frequency (cdf) of the RDS instances for the lifetime given by the X coordinate. A common behavior across almost all benchmarks is that at least one of the RDS instances tend to be alive almost throughout the program. This is evident from the fact that when the cdf reaches a value of 100%, the X co-ordinate is close to 100%. This conveys the fact that programs tend to have one “core” RDS that is created during the initialization phase and is live almost till the end. Another view of the distribution of the RDS instances over time is given by Figure 6. In this figure we plot the normalized life time in the X axis and the number of live RDS

Benchmark	# Dynamic Instructions *in billions*	Time (Baseline) *in secs*	Time (with Profiling) *in secs*	Memory Usage *in MB*
130.li	0.65	12	137	< 1
175.vpr	57.83	652	11295	1.5
188.ammp	102.8	3538	22171	3.5
197.parser	24.9	276	9377	122
253.perlbmk	105.9	2445	32221	85
olden_bh	2.51	28	170	< 1
olden_mst	0.56	5	113	88
ks	.02	3	10	< 1
tree_puzzle	163	1447	19126	152.6

Table 2: Execution time and space requirement

Benchmark	#RDS Types	#RDS Instances	#Fwd. Edges	#Bkwd. Edges	#Avg. Size	#Avg. Lifetime (normalized)	Total Accesses	Avg. Latency
olden_bh	2	5	1666	511	435	98.26	130175	1.86
olden_mst	1	2048	0	14208	6	47.27	32117	2.77
130.li	1	1	2697460	561356	3258816	99.99	9678408	3.67488
175.vpr	2	877	4742	0	5	0.121	28821	4.45
188.ammp	7	8	3723951	16027	467497	95.7713	636186339	4.14577
197.parser	31	1409099	28533225	37991142	47	0.28	707958303	3.92
253.perlbmk	4	29	520	236	26	24.12	26156678	1.00568
ks	3	646	14155	14385	44	99.9	1480740810	1.07221
tree_puzzle	3	3	36	31	22	57.01	527833	1.30975

Table 3: Characteristics of RDS

instances in the Y axis. At time 0, the number of RDS instances is 0. In most of the benchmarks, the number of RDS instances reaches a non-zero value soon and remains non-zero almost till the end of program execution. This does not contradict our hypothesis that there is at least one RDS instance that is created early and remains alive till the end. Another type of interesting behavior is shown by *197.parser*. This benchmark has the maximum number of RDS instances among all the benchmarks we have profiled. In Figure 5, the line for parser shows a steep increase immediately after time 0, and stays slightly less than 100 almost near the end. This implies that an overwhelming fraction of the RDS instances have very short normalized lifetimes, but there is at least one instance which is alive for almost the entire life of the program. These observations match well with the actual behavior of the benchmark as seen from its source code. The application uses RDS to first create a dictionary. Then, as it reads the input file, it creates a bunch of data structures for each sentence and parses the sentence. Once the sentence is parsed, it deletes the RDS instances corresponding to that sentence. These RDS instances created for each of the sentences are the short living RDS instances, while the RDS created for the dictionary is alive throughout the entire program.

6.2.2 RDS stability factor

As stated in the previous section, we use the RDS stability factor (RSF) metric to quantify the stability of the RDS. In this section, we show how stable are the RDS instances in our benchmarks. Figure 7 shows the cdf of the RSF. We plot the X axis (RSF) only up to a value of 10. The Y axis shows the percentage of RDS instances weighted by the pointer chasing loads within the given RSF. We find that in many benchmarks, most pointer chasing loads belong to RDS instances that have good RSF values ($<= 2$). On the other side of the spectrum, *188.ammp* has a negligible fraction of loads within a RSF of 10, and in *197.parser*, only about 35% of them have a RSF within 2. In case of *188.ammp*, the major fraction of

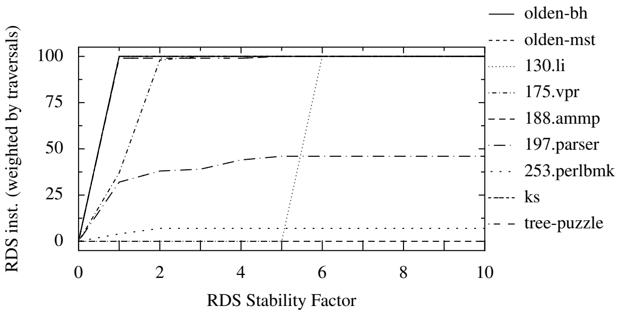


Figure 7: Cumulative distribution of RDS stability factor

the pointer chasing loads occur in two lists : a list of *atom*s and a list of *tether*s. The program reads an input file, sometimes adds new elements to one of these lists, and traverses the lists in between. Thus the lists keep expanding as the input is read and hence the traversals get distributed across several alteration points. On the other hand, *Olden* benchmarks typically create some data structures and then process them, thereby having a good RSF value.

7. CONCLUSION AND FUTURE WORK

In this paper, we introduce a new profiling technique called shape profiling. We describe how shape profiling identifies the logically disjoint recursive data structure instances in a program, without requiring a high level program representation or type information of program variables. Using shape profiling, we were able to identify various properties of RDS in a set of benchmarks that are not revealed by other profiling techniques. We also describe the notion of stability of a shape and define a metric to quantify it. Our implementation of the profiler had a manageable time and space overhead.

The future work includes leveraging this technique to capture more interesting properties of shapes. We plan to investigate compiler optimization techniques that could use this shape profile information to optimize at the granularity of data structure instances.

8. REFERENCES

[1] CALDER, B., KRINTZ, C., JOHN, S., AND AUSTIN, T. Cache-conscious data placement. In *Proceedings of the 8th International Symposium on Architectural Support for Programming Languages and Operating Systems ASPLOS'98* (October 1998).

[2] CLARK, D. W. *List structure: measurements, algorithms, and encodings*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1976.

[3] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press and McGraw-Hill, 1992.

[4] GHIYA, R., AND HENDREN, L. J. Is it a tree, dag, or cyclic graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages* (January 1996).

[5] HACKETT, B., AND RUGINA, R. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), pp. 310–323.

[6] KAPLAN, H., SHAFRIR, N., AND TARJAN, R. E. Union-find with deletions. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2002), pp. 19–28.

[7] LATTNER, C., AND ADVE, V. Automatic pool allocation for disjoint data structures. In *Proceedings of the Workshop on Memory System Performance* (2002), ACM Press, pp. 13–24.

[8] LATTNER, C., AND ADVE, V. Data structure analysis: A fast and scalable context-sensitive heap analysis. Tech. Rep. UIUCDCS-R-2003-2340, University of Illinois, Urbana, Illinois, April 2003.

[9] 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 *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation* (June 2005).

[10] LUK, C.-K., AND MOWRY, T. C. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *Proceedings of the 26th International Symposium on Computer Architecture* (July 1999).

[11] NYSTROM, E. M., JU, R. D., AND HWU, W. W. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the 28th International Symposium on Computer Architecture* (September 2001).

[12] PEARCE, D. J., AND KELLY, P. H. J. Online algorithms for topological order and strongly connected components. Tech. rep., Imperial College, September 2003.

[13] RUGINA, R. Quantitative shape analysis. In *Proceedings of the 11th Static Analysis Symposium* (2004).

[14] SAGIV, M., REPS, T., AND R.WILHELM. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (January 1996), pp. 16–31.

[15] VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J. A., AND AUGUST, D. I. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)* (November 2002), pp. 271–282.

[16] WU, Q., PYATAKOV, A., SPIRIDONOV, A. N., RAMAN, E., CLARK, D. W., AND AUGUST, D. I. Exposing memory access regularities using object-relative memory profiling. In *Proceedings of the International Symposium on Code Generation and Optimization* (2004), IEEE Computer Society.