How Successful Is Data Structure Analysis in Isolating and Analyzing Linked Data Structures?

Patrick Meredith, Balpreet Pankaj, Swarup Sahoo, Chris Lattner and Vikram Adve
Computer Science Department Technical Report #UIUCDCS-R-2005-2658
University of Illinois at Urbana-Champaign
{pmeredit,bpankaj2,ssahoo2,lattner,vadve}@cs.uiuc.edu
November 2005

Abstract

This report describes a set of experiments to evaluate qualitatively the effectiveness of Data Structure Analysis (DSA) in identifying properties of a program's data structures. We manually inspected several benchmarks to identify linked data structures and their properties, and compared these against the results produced by DSA. The properties we considered are those that were the primary goals of DSA: distinguishing different *kinds* of data structures, distinct *instances* of a particular kind, *type* information for objects within an LDS, and information about the *lifetime* of such objects (particularly, those local to a function rather than global). We define a set of metrics for the DS graphs computed by DSA that we use to summarize our results concisely for each benchmark. The results of the study are summarized in the last section.

1 Introduction

The algorithm called Data Structure Analysis (DSA) by Lattner and Adve [4] is designed to identify and isolate instances of linked data structures (LDS), their lifetimes, and internal structural and type information, but not to prove shape properties. In particular, our goal has been to develop an algorithm that is very fast in practice (sacrificing analysis power where necessary) and also is designed to handle challenging practical issues faced by production compilers, including incomplete and non-type-safe programs, reuse of data structure manipulation functions for separate LDS instances, and complex features of C and C++.

Because our algorithm trades off analysis precision for speed in several important ways, it is important to evaluate specifically how successful the algorithm is in extracting the relevant properties of data structures. The four major properties of data structures DSA aims to extract include:

- distinct *kinds* of data structures;
- distinct *instances* of a particular kind;
- *type* information for objects within a linked data structure; and

- the *lifetime* of such objects (particularly, to distinguish objects that are local to a function rather than global).

Unfortunately, evaluating *quantitatively* whether DSA actually identified "logical data structures" in programs is difficult for two reasons:

- First, the notion of what is a data structure is somewhat subjective. Complex, pointer-based data structures are hierarchical, e.g., an array of trees of linked-lists. Furthermore, many sub-structures can be interconnected for incidental reasons that may be best interpreted as separate data structures if they serve different purposes, e.g., when two unrelated entities like a tree and a hash-table are pointed to by a common global container structure for programming convenience.

- Second, we do not have an alternative tool that can analyze the relevant properties of data structures. This makes it difficult to systematically identify the data structures in order to compare how well the algorithm does.

To evaluate the algorithm, therefore, we have performed a painstaking *manual but partially qualitative* experiment. We manually inspected several of our benchmarks to identify linked data structures and their properties, and compared these against the results produced by DSA. To address the problem of hierarchical organization above, we separately considered instances of *linked data structures* and instances of identifiable sub-data-structures within the larger structures (which we call *logical data structures*). These terms are defined concretely in Section 2.3.

This technical report presents the detailed results of this manual experimental study. The next section describes the goals of the study, the terminology we use to make these goals and our metrics concrete, and a set of metrics we define to quantify the key data structure properties in which we are interested. Section 3 describes our experimental methodology and the five benchmarks we use in this work. The following five sections describe the detailed results for each of these five benchmarks. Finally, Section 10 summarizes the conclusions we arrive at in this work.

2 Goals and Metrics

As noted in the Introduction, the broad goal of our experiments is to evaluate how successful the DSA algorithm is in extracting the properties of data structures it aims to extract: disjoint instances, lifetime, and type information.

2.1 Goals of the Study

The questions (corresponding to key data structure properties) we considered in this evaluation include:

1. Was DSA successful at distinguishing linked data structures (LDS) of *different kinds* (ignoring distinct instances of a particular kind)? For example, a program that creates two similar linked lists and three similar binary trees, would DSA distinguish the lists from the trees? A simple type-based analysis is sufficient in a strongly typed language but not in C. This requires sufficiently precise memory disambiguation so that objects of different types are not incorrectly aliased, and

requires inferring accurate type information from memory operations (e.g., even in the presence of casts) so that the field-sensitive analysis can capture the internal structure of each kind of data structure.

2. Was DSA successful at distinguishing *disjoint instances* of LDS of a particular kind? Interestingly, this question includes three distinct cases that require different analysis capabilities:

   (a) Distinct instances created at two different places in the program, whether in the same function or different function.

   (b) Distinct instances created and used within different invocations of the same function (i.e., local to a function).

   (c) Distinct instances created (typically at one place, within a loop or recursion) and stored into another aggregate structure (e.g., an array or list of pointers to the instances).

   These tend to have different outcomes because the first one requires context-sensitivity, the second requires accurate lifetime information, and the third requires flow-sensitivity plus some potentially more sophisticated analysis such as shape analysis (e.g., for a list of disjoint lists), or array dataflow analysis and dependence analysis (e.g., for an array of disjoint lists).

3. For data structures whose lifetime is local to a function, was DSA successful at proving the lifetime is local? More broadly, how many data structures were identified as local vs. global? The latter directly determines the answer to question 2(b) above, by focusing on functions that are potentially invoked more than once.

4. Was DSA successful at extracting type information for individual DS nodes within the data structures? This information enables our field-sensitive analysis to track distinct pointer fields and therefore determines whether DSA is able to extract structural information about each kind of data structure. For efficiency, DSA is field-sensitive only for DS nodes where all referenced fields of all objects have compatible types.

2.2 Strengths and Weaknesses of DSA

Understanding the outcomes of these questions requires an understanding of the basic strengths and weakness of the DSA algorithm. We briefly summarize these as follows; they are explained in more detail in [6, 4]:

Strengths

1. *Type information*: Ability to infer type information for sets of objects (DS nodes). Where available, this information also enables field-sensitive analysis, i.e., tracking of distinct pointer fields.

2. *Context-sensitivity*: Ability to distinguish heap objects via acyclic call paths. This enables distinguishing two or more instances in case 2(a) above, and also makes it more likely in case 2(b). By allowing more precise memory disambiguation, it can also make it more likely to extract precise type information or lifetimes.

3. *Escape information*: Ability to distinguish lifetimes of sets of objects (DS nodes) via reachability in the points-to graph. This enables distinguishing two or more instances in case 2(b) above.

Weaknesses

1. *Flow-insensitivity*: The lack of flow-insensitivity in the basic alias analysis algorithm can lead to imprecision in all the goals above, especially when a single a pointer is used at different places in the program to point to different objects with different properties (e.g., local vs. global lifetimes, different types, or distinct instances). We refer to this as *basic flow insensitivity*.

   Static analysis algorithms for data structures can conceivably have more powerful capabilities that require flow insensitivity together with additional analysis features. Two important ones we identify (based partly on our experience in this study) as lacking in DSA are:

   - *No uniqueness information*: We use this term informally to refer to the key "#1" property exploited by previous shape analysis algorithms [7, 2]. A static representation of a set of dynamic memory objects (e.g., a DS node) has this property at a program point if every dynamic memory object is guaranteed to have at most one incoming pointer from any other memory object at that program point.
   - *No array dataflow analysis*: Any algorithm that can distinguish data structures instances reached from two distinct array elements (in an array of pointers) would generally require some form of array dataflow analysis.

2. *Unification*: DSA may merge two sets of objects if a particular pointer variable or field may point to objects in either set. Like flow-sensitivity, unification can lead to imprecision in all the goals above.

3. *Limited context sensitivity in recursion*: DSA cannot distinguish distinct heap objects created in different function invocations within a recursive computation if those objects in certain cases. For example, if objects created in one invocation are passed as arguments to a later recursive invocation (the incoming arguments to the recursive function would be merged with the outgoing arguments to the recursive call). This is true with either self-recursion or mutual recursion. On the other hand, DSA *can* distinguish heap objects created within an invocation and only used locally or passed to (or returned from) only callee functions outside the strongly connected component of the call graph. We will refer to these as *recursive argument objects* and *non-recursive objects* respectively.

2.3 Terminology

To make our experiments and metrics concrete, we use the following terms and definitions. The first two terms are defined with respect to data types and usage within the program and are unrelated to the analysis.

Data Structure Snapshot (DSS) A set of memory objects at runtime that form nodes of a connected graph, where an edge is formed by a pointer from one object element to another, and elements of individual objects can be traversed via structure or array indexing. This gives the key property that a DSS must be *traversable* via sequences of pointer dereferences and indexing operations. One DSS can contain another; for example, an array of binary trees forms a DSS, and so does each individual tree (viewed at a particular point in an execution).

Linked Data Structure (LDS) : A maximal subset of user-defined data types in a program such that there is some DSS in some execution that consists of instances of exactly these types and a pointer edge from field $O_1.f_1$ to $O_2.f_2$ in the DSS corresponds to valid fields f_1 and f_2 for the static types of O_1 and O_2.

This gives a precise definition for the static notion of a data structure. Defining it in terms of dynamic snapshots correctly captures the case of a generic data type (e.g., a list of `void*` used for lists of two different kinds of data types). It is also independent of any particular analysis.

Unfortunately, this definition by itself is not very useful in practice because it is common for many intuitively distinguishable "logical data structures" to be grouped into larger structures for convenience, as noted in the Introduction. The next term captures these smaller, logical structures, but it is less precise and identifying them requires some subjective judgement and we do it manually.

Logical Data Structure (Logical DS) : A minimal subset of user-defined data types within an LDS that intuitively serve a single purpose within the input program.

For example, if a program creates a hash table mapping identifiers to binary trees (and no other data types point to the hash table or trees), the hash table with the trees would form a linked DS, while the data types forming the hash table and those forming the tree would be two separate logical DSs. Note that the tree may contain multiple node types and may still be a single logical DS.

On the other hand, we often consider a small hierarchy, e.g., an array of trees or a list of lists, as a single logical DS if the array or the list of lists is the primary focus of the algorithm. Distinguishing these two kinds of situations is why we consider this step subjective.

Note that the only subjective step here is to identify a set of types that intuitively serve a "single purpose" within the program. Once that set has been identified, the remaining definitions and metrics used in the study are precise.

To evaluate the properties of the static analysis, we must be able to refer to *sets of instances* of LDSs and logical DSs distinguished by the analysis. For example, a single logical DS consisting of a linked list data type may be represented by multiple DS nodes (clones of each other) in different functions or within the same function. Two distinct DS nodes in the same or different functions that are "complete" represent disjoint lists being created in the program. Each "complete" DS node has also been proved local, so it proves that distinct dynamic instances of the logical DS are created at run-time in separate invocations of the function. Both these situations represent multiple instances of a single logical DS. We refer to each such DS node (in this example) or set of DS nodes as a "static data structure instance" of the single logical DS.

Static Data Structure Instance (SDSI) : A sub-graph of a function's DS graph whose nodes represent instances of the data types in a logical DS, and whose edges correctly capture a superset of the points-to properties between those instances, and such that every DS node in the subgraph is complete (no I flags).

Completeness is important for two reasons: (a) to count an SDSI only once, instead of in every function to which it is passed or from which it is returned; and (b) to ensure that any potential collapsing of type information has been considered.

Static Collection : An SDSI containing an Array node (the **A** flag is set) or a cycle (including a self-cycle), or a set of SDSI containing a cycle. Any *other* SDSI reachable from a static collection may have multiple instances at run-time reachable from a single instance of the collection. This helps identify case 2(c) of distinct instances discussed earlier, e.g., an array or list or tree of SDSIs.

2.4 Metrics for Data Structure Properties

In order to summarize the results of our study concisely for each benchmark, we use the following metrics:

N $\equiv$ Total number of SDSI in all function DS graphs.

G $\equiv$ Number of SDSI that are global, i.e., some node in the SDSI is reachable from a DS node marked **G**.

L $\equiv$ Number of SDSI that are not global. Since nodes in an SDSI are complete, this means that they are local to a function. By definition, $L = N - G$.

K $\equiv$ Total number of **H** nodes in all SDSI.

I $\equiv$ Total number of **H** nodes in all SDSI that satisfy the following property: there are two or more **H** nodes in a function's DS graph have identical types representing disjoint instances in the same function and no instance is Collapsed.

O $\equiv$ Number of **H** nodes in all SDSI that are Collapsed.

C $\equiv$ Number of **H** nodes in all SDSI that are reached by at least one Collection (which may be the node itself).

Note that N, G and L refer to static instances of logical DSs (i.e., SDSI), whereas K, I, O and C refer to DS nodes. An SDSI may contain multiple DS nodes. I identifies occurrences of case 2(a). L identifies occurrences of case 2(b), and O identifies occurrences of case 2(c).

3 Methodology and Benchmarks

3.1 Benchmarks Used in the Study

Because of the time and effort required, we are only able to do this study for a small number of moderate size programs (smaller programs usually have few interesting data structures and often only one instance of each). We chose moderate-size, pointer-intensive programs that do not use custom allocators (except for parser, discussed below). Three are from SpecInt2000: `175.vpr`, `197.parser-b`, `300.twolf`. The three other programs are `fpgrowth` (a small program that has interesting recursive behavior with respect to creation of data structures, `espresso` and `eon`.

Although `197.parser` uses a custom memory allocator, the interface to this allocator is semantically equivalent to `malloc/free`, but this cannot be inferred even with "full" context-sensitivity (i.e., they are not just malloc "wrappers"). In the original program, virtually all interesting heap objects are merged into a single DS node because of the custom allocator. We changed the allocation routines in 197.parser to call malloc/free directly (now context-sensitivity is enough!), yielding the `197.parser(b)` benchmark. Because it calls malloc & free, DSA is able to identify many interesting logical data structures.

3.2 Methodology

Briefly, we took the following steps for each of the benchmarks:

1. Run DSA to construct the "Complete Bottom-Up" (CBU) DS graphs. We use the CBU graphs because these are the most complete "summary" information for the side-effects of a function, including the side-effects of all its known callees. This graph is the starting point for client queries, including:

   - *Alias analysis*: The CBU graphs are inlined top-down to construct the TD graph for each function, which includes aliasing information for all callers. Technically, this top-down inlining can be performed with either the BU or CBU graphs with identical results, so we actually do it with the BU graphs. Nevertheless, the conceptual effect is like using the CBU graphs.
   - *Inter-procedural (IP) Mod/Ref*: The CBU graphs of a subset of callees is inlined into the TD graph of a caller at a particular call site to compute the IP Mod/Ref side-effects for those callees at that call site.
   - *Automatic Pool Allocation*: This transformation takes the CBU graph as its primary input to insert pool descriptors and identify their lifetimes [5].

2. We then identified all the "Complete" nodes with 'H' or 'S' markers in order to identify nodes that are potential members of an SDSI.

3. Using these nodes, we examined the data structures in the program source to understand many properties of these data structures, including their data types, lifetimes (including identifying DSs with globals pointing to them), and the functions where they were created and used. Although we studied many such properties, the specific question we had to answer for each function was which subsets of the above complete DS nodes should be considered a single logical DS? This is the subjective step mentioned earlier.

4. Identify the DS nodes corresponding to static instances of each such logical DS. This identifies the SDSI in each function. For example, there are two static instances of the Connector struct logical DS in Figure 11 and also in Figure 13.

5. Count the SDSI in all functions (N) and the cases of SDSI or SDSI nodes matching the other metrics listed previously. One key question we must answer in this counting is this: When two SDSI within a function are identical in structure, do they correspond to disjoint instances of a single logical DS? And do they require context-sensitivity with heap cloning (they do not if the actual allocation sites

are distinct for the two SDSI). In all cases, we have encountered, the answer to both these questions is "*Yes*".

4 Results for fpgrowth

4.1 Summary:

This is a specific implementation of fpgrowth algorithm [3], which is one of the most efficient algorithms to mine frequent-itemset using divide-and-conquer strategy. This algorithm uses many instances of special tree data structures called FP-tree. This implementation has 634 lines of code, 544 memory instructions in LLVM representation and 1 strongly connected component in the call graph.

4.2 Data Structures:

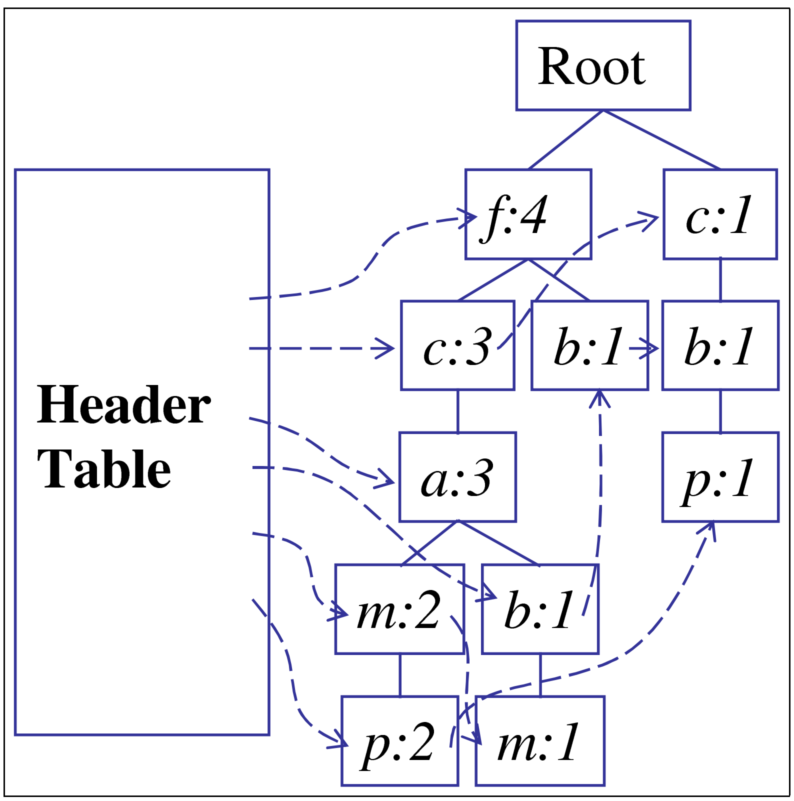

Figure 1: Structure of FP-tree data structure (taken from Shengnan Cong

There are two significant kinds of data structures in this implementation - *FPtree* and *large item table*. The structure of the main data structure fptree is shown in Figure 1. It has a root node and a set of nodes forming a tree structure. The tree structure (shown with solid lines in the figure) is imposed using three pointer variables. "Parent" variable points to the parent of the given node, "first_son" variable points to the first child, "right_brother" points to next sibling. The header table contains pointers to first node with a particular label. The "next" pointer in each node forms a linked list of nodes with same label (shown with dotted arrows in the figure). The large item table data structure contains number of large items, an array of large items and their count (i.e number of occurrence of each large item).

One main FPtree is created in build_fp_tree_from_transaction function. The main function calls a recursive function fptree_mine. A local FPtree is created in this function and passed as an argument to the recursive call. Main FPtree is passed to the first call from main function. In this way many local FPtree data structures are created in fptree_mine function. In a similar manner, a large item table is passed as an argument to the fptree_mine function, where many local large item tables are created locally. Another set of local data structures called conditional table are also created recursively in this function. But it is not passed on to the recursive call as an argument.

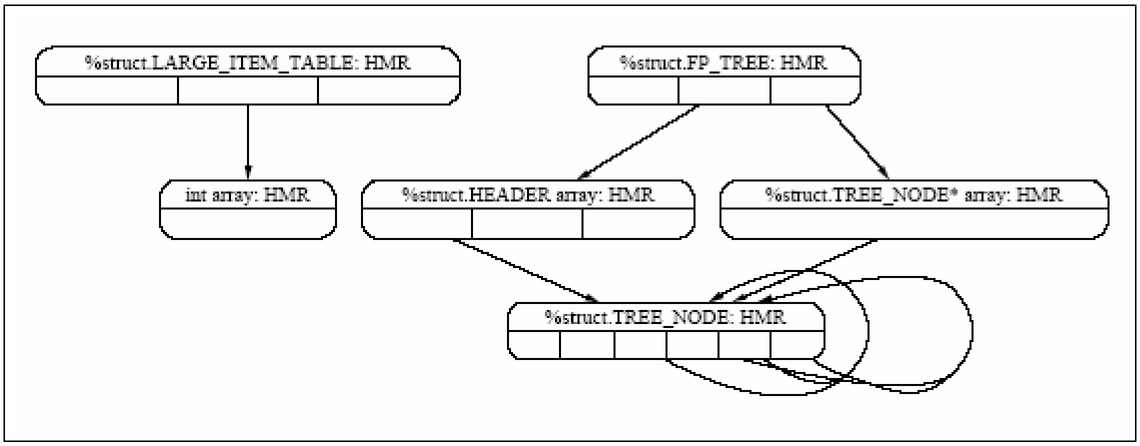

Figure 2: CBU graphs of main data structures in fpgrowth

The part of CBU graph of main function showing the parts corresponding to the important data structures is shown in Figure 2. Fptree data structure contains tree_node structure instances forming the tree. The figure also shows header array, which forms a linked list of tree_node having same label. This figure also shows the structure of the large item table in main function. These data structures are local to main. DSA correctly identifies them as local, as evident from the absence of "G" flag. The nodes also contain accurate type information without being collapsed. But the DSA is unable to find disjoint instances of main and local fptree data structures due to limited context sensitivity in recursion. This is also the case with large item table, as it is also a recursive argument object like FPtree. But DSA identifies disjoint local conditional table data structures, as they are non-recursive objects (i.e. not passed as argument to recursive calls).

4.3 Metrics:

The values of the described metrics for fpgrowth are -

N=3

Three SDSI were found in this code. The data structures include one fptree, one large item table and one conditional table data structure.

G=0

No SDSI was global in this case.

L=3

All data structures were correctly proved to be local by DSA.

K=9

The main data structures consist of 9 nodes.
I=0
There are no disjoint instances/nodes of same type.
O=0
None of the nodes are collapsed.
C=2
These two nodes form parts of two data structures.

5 Results for espresso

5.1 Summary:

Espresso is an integer benchmark. The goal of espresso benchmark is to minimize boolean functions using set operations such as union, intersect and difference. The input to the benchmark is a boolean function and output is a logically equivalent function possibly with fewer terms. The implementation consists of 14959 lines of code and approximately 50 source files.

5.2 Data Structures:

There are two important data structures in this implementation. First, there are disjoint symbolic_struct structures, symbolic_label_struct structures and disjoint symbolic_list_struct structures pointed to by program variables. The part of CBU graph showing some parts of these data structures is shown in Figure 3. The disjointness is detected by context sensitivity in DSA algorithm. The significant part of this data structure consists of linked list of linked lists. This data structure is local to main function and DSA correctly identifies this using the fact that none of nodes marked with "G" flag point to these structures. None of the nodes are collapsed and all the nodes have accurate type information.

Another interesting data structure is a sparse matrix representation in do_sm_minimum_cover function. The part of CBU graph showing the parts of this data structures is shown in Figure 4. the rows and columns form doubly linked lists in this data structure. All the matrix elements are connected in a mesh form, where each element has pointers to its four neighbors along the same row and column. DSA algorithm correctly identifies disjoint arrays of sm_col_struct structures using context sensitivity. DSA also finds disjoint local sparse matrix structures using escape analysis. This is correctly identified as a local data structures as they are not reachable from any node marked "G". DSA was able to infer the correct types for all nodes and none of the nodes got collapsed. For example, DSA incorrectly shows pointers to sm_col_struct in stead of sm_row_struct.

5.3 Metrics:

The values of the described metrics for this benchmark are - N=22
The main data structures found include linked list of symbolic_struct, linked list of linked of symbolic_list_struct and symbolic_label_struct structures, disjoint pair_struct structures, few disjoint sparse matrices, disjoint set_family structures, few disjoint solution_struct, sm_col_struct, sm_element_struct

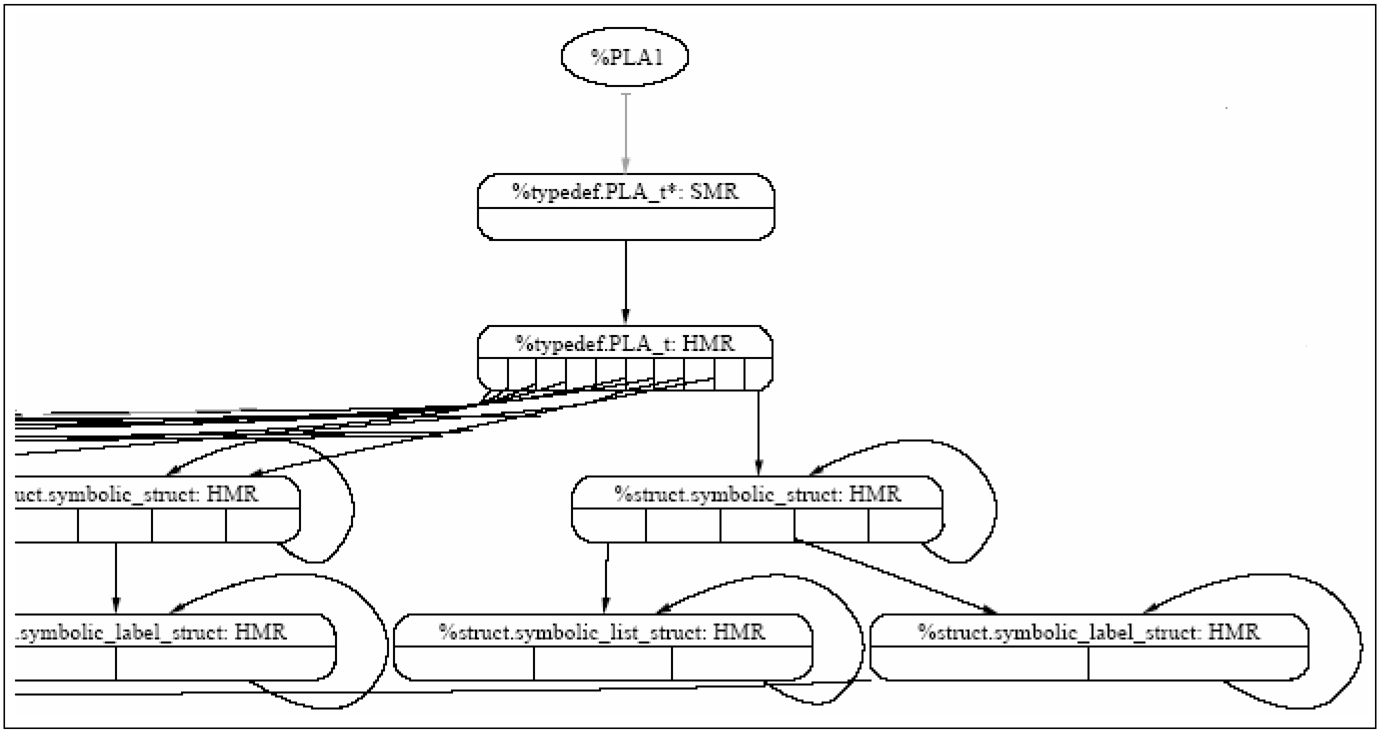

Figure 3: CBU graphs of linked list of linked lists in espresso

structures etc.

G=8

L=14

All local and global nodes were identified correctly

K=148

There were large number of heap nodes in this case

I=67

Many disjoint "'H"' nodes were found, which signifies the large number of disjoint data structures identified by DSA algorithm.

O=0

None of the nodes were collapsed in this case.

C=47

Large number of nodes are reachable from a collection. This provides a hint on the large number of non-trivial data structures found by DSA in espresso.

6 Results for 300.twolf

6.1 Summary:

This is a part of the SPEC INT 2000 Benchmark known as TimberWolfSC placement and global routing package. This benchmark determines the placement and global connections for groups of transistors which constitute a microchip. The placement problem is a permutation. In stead of a simple brute

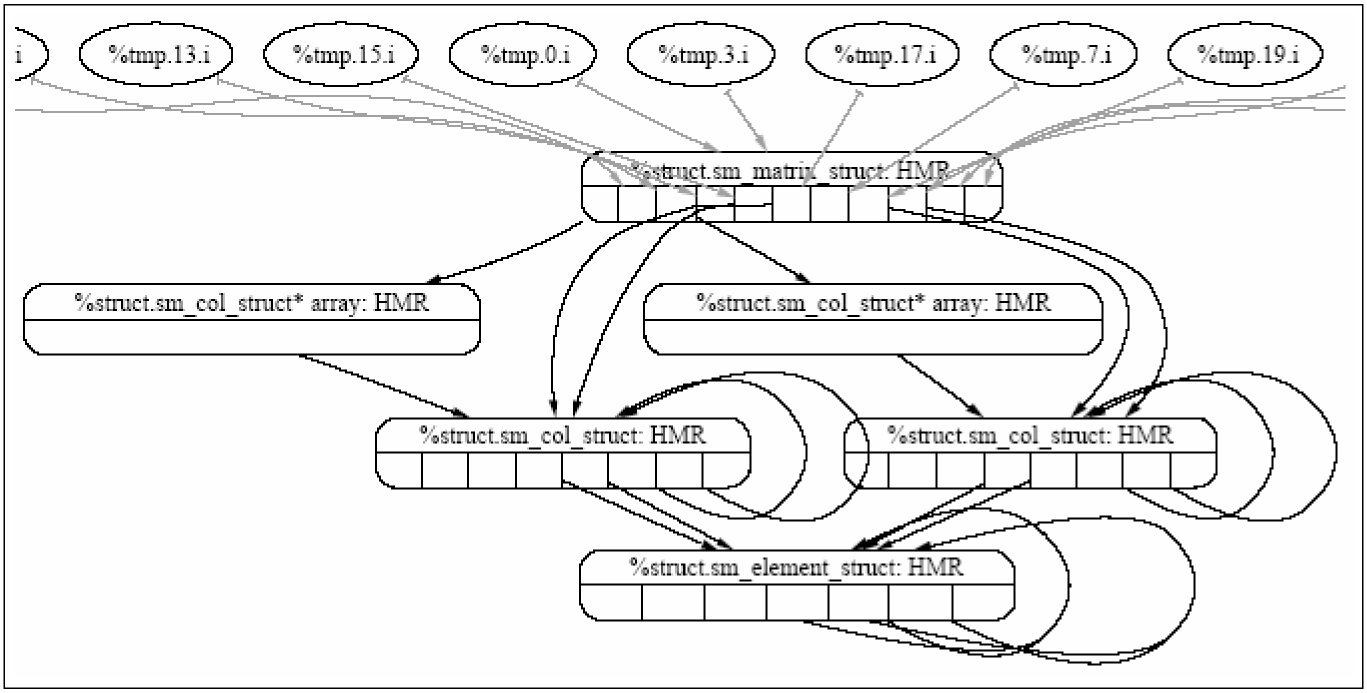

Figure 4: CBU graphs of sparse matrix in espresso

force exploration of the state space, the TimberWolfSC program uses simulated annealing as a heuristic to find good solutions. This benchmark has 20459 lines of code, 19686 memory instructions in LLVM representation and 1 non-trivial strongly connected component in the call graph.

6.2 Data Structures:

There are a few important data structures in this implementation. First, there are disjoint arrays of blockbox structure created in "configure" function. The part of CBU graph showing the parts corresponding to this data structure is shown in Figure 5. Two pointer variables "barray" and "oldbarray" points to disjoint blockbox arrays in some parts of code, but in the cbu graph they point to same array as oldbarray points to barray at one place in the code. Flow sensitivity or subset based pointer analysis is required identify to disjoint nodes in this case. This data structure is global and DSA correctly identifies this using the fact that the arrays are reachable from nodes marked with "G" flag. The nodes also contain accurate type information without being collapsed.

Another interesting data structure involves linked list of ibox and ipbox structures created in buildimp, build_feed_imp functions. The part of CBU graph showing the parts corresponding to these data structures is shown in Figure 6. As can be seen from the figure, this data structure is actually a linked list of linked list. These are global data structures as they are reachable from nodes marked "G". The are two distinct instances of ipbox structure in build_feed_imp function. But DSA was unable to detect it due to unification. Each element of impFeeds array points to distinct ibox linked list and each ibox instance points to distinct ipbox linked list. DSA is also unable to identify this feature due to lack of flow sensitivity and array dataflow analysis as mentioned earlier. The nodes also retain their correct types without being collapsed.

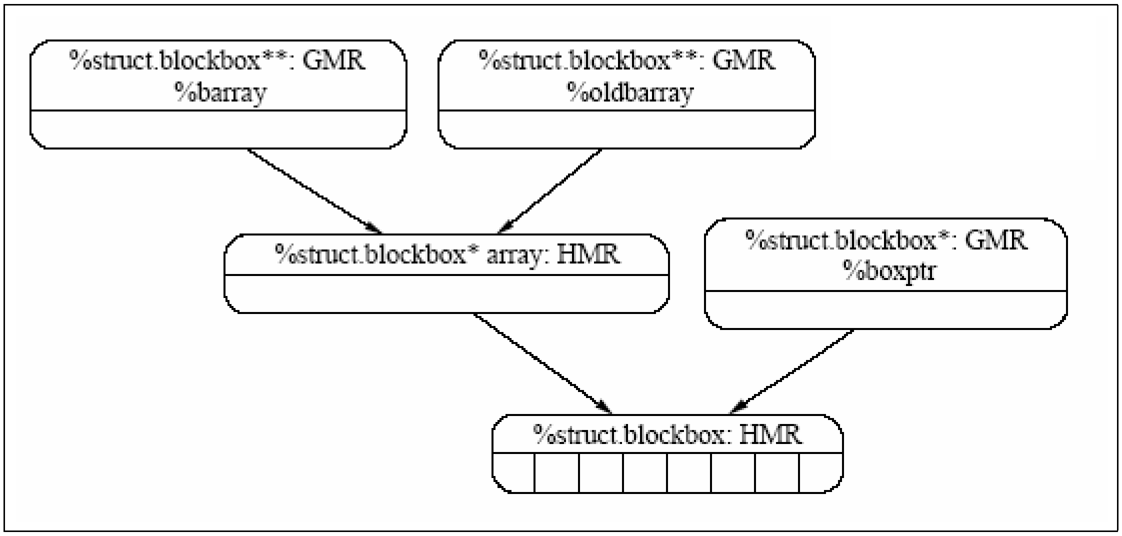

Figure 5: CBU graphs of blockbox array in twolf

Third interesting data structure is actually a part of a complex data structure involving many structures. The main component of this data structure consists of linked lists of termbox and netbox. "carray" and "tearray" both points to the linked list of netbox. The CBU graph containing the relevant parts is shown in Figure 7. These linked lists are also global data structures as they are accessible from global variables. As before DSA correctly identifies this. Also, all the nodes have been assigned correct types without any collapsed node.

Another significant data structure is also a part of complex data structure. The interesting aspect of this data structure involves doubly linked lists of changridbox and densitybox data structure. They also contain a pointer to each other. Tgrid, Shuffle, aNetSeg all contain pointer to the doubly linked list of changrdbox. The corresponding CBU graph is shown in Figure 8. DSA correctly detects these as global data structures. All the nodes also have correct type information except one node, which is collapsed.

6.3 Metrics:

The values of the described metrics for this benchmark are - N=23

The main SDSIs include arrays of blockbox structure, linked list of ibox and ipbox structures, linked lists of termbox and netbox, doubly linked lists of changridbox and densitybox, array of dimbox, cellbox, segbox, tgridbox, densitybox, binbox and rowbox structures, a hash table.

G=21

twenty one global nodes were identified.

L=2

Both the local nodes were identified correctly.

K= 97

The code had lots of heap allocated objects.

I=6

Few disjoint "'H"' nodes were found by DSA.

O=7

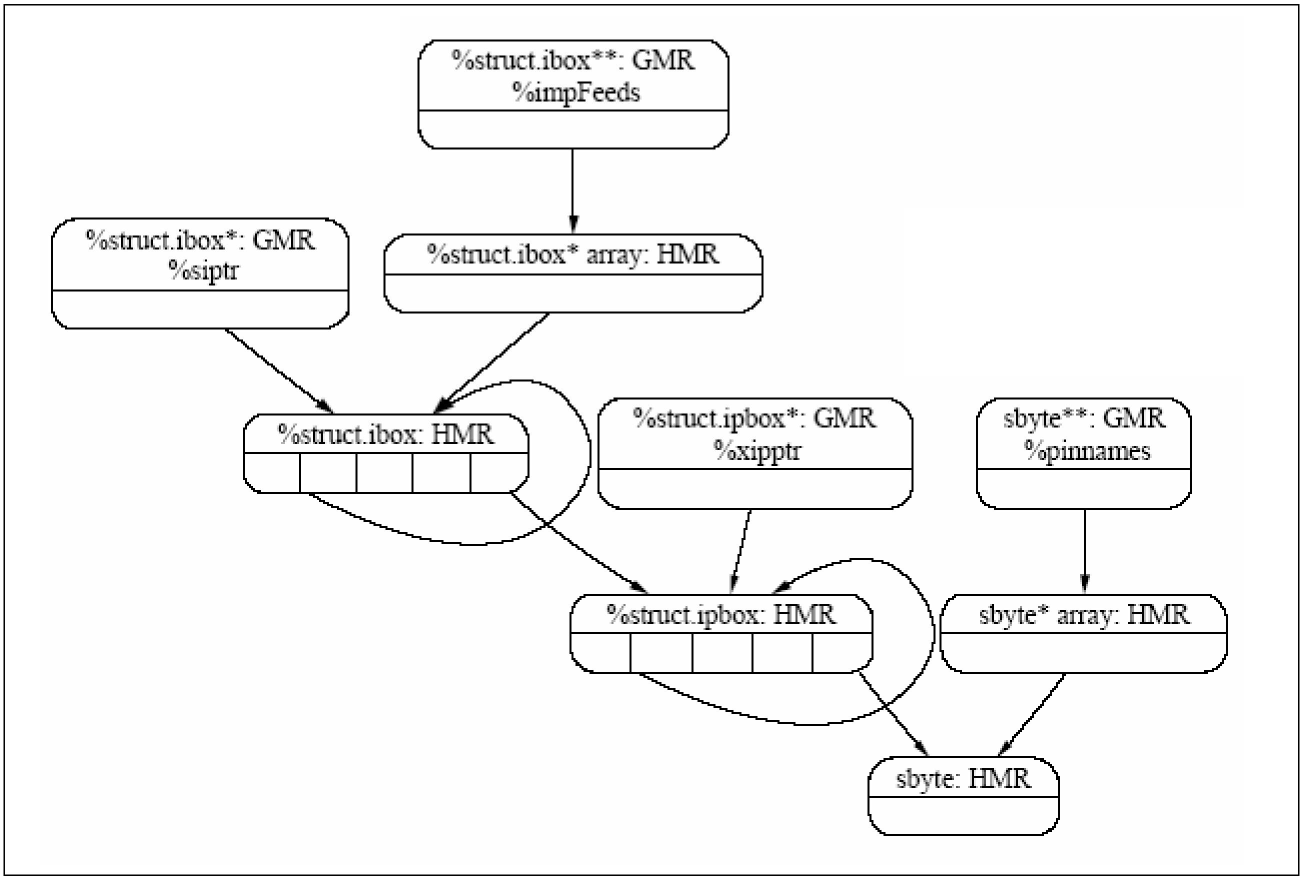

Figure 6: CBU graphs of ibox and ipbox linked lists in twolf

Few nodes got collapsed in this case.
C=29
All these nodes form a part of the non-trivial data structures found.

7 Results for 181.mcf

Summary:

The 181.mcf is a part of the SPECInt2000 benchmark. It is a benchmark derived from a program used for single depot vehicle scheduling problem in public mass transportation. The code is written in C and uses almost exclusively integer arithmetic.181.mcf has a raw LOC of 2412 with a total of 991 memory instructions in LLVM representation.

Data Structures:

There are two major data structures created in the benchmark:
i). Network : Models the complete flow network
ii). Basket : Contains the array of sorted edges

Both these data structures are global and have a lifetime throughout the code. The network data structure is initialized in the function readmin() called from main. The function readmin() reads the

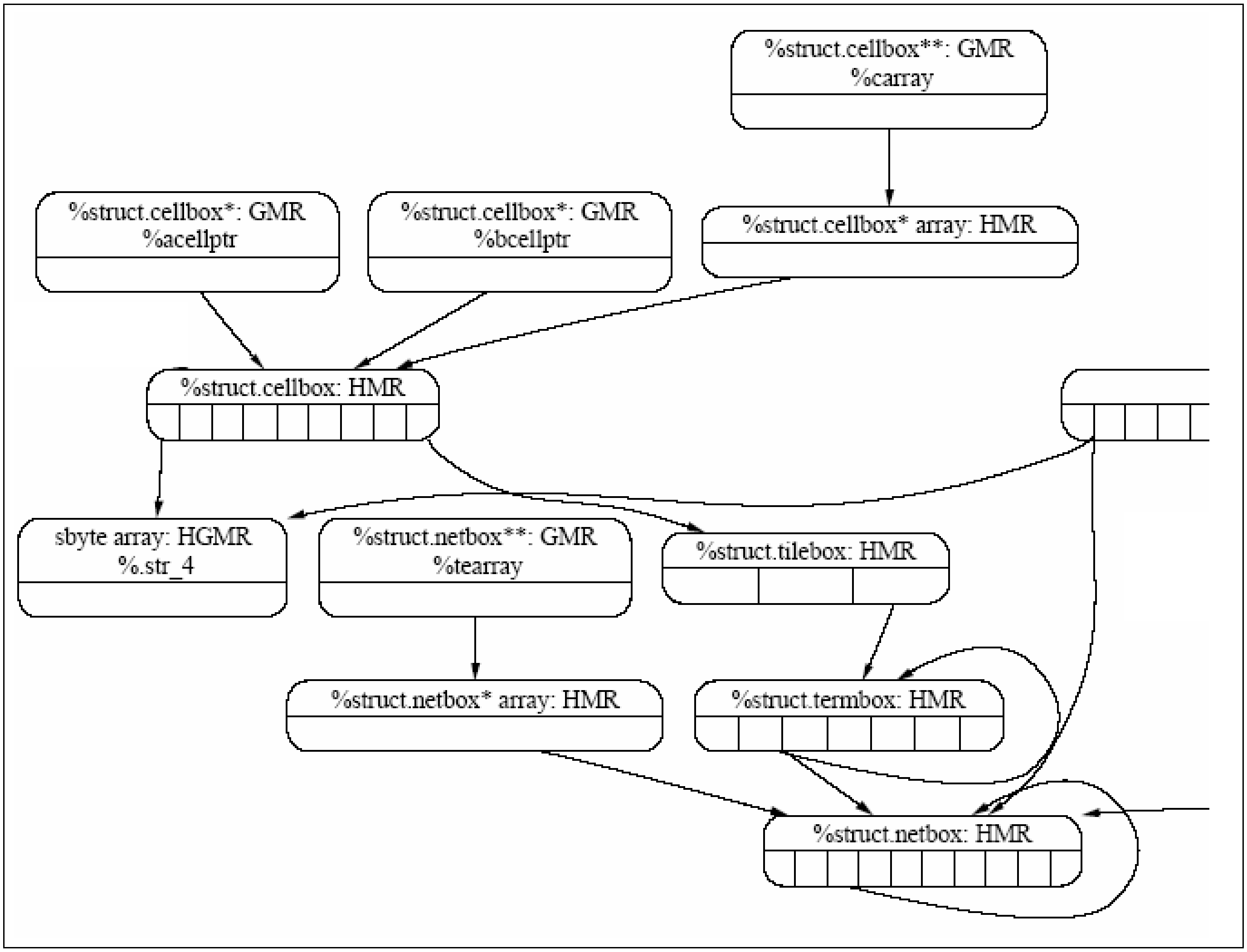

Figure 7: CBU graphs of termbox and netbox linked lists in twolf

complete network graph from an input file and models this as a data structure.

Figure 9 shows the cbu (complete bottom-up) graph for the main function. The network and basket data structures are clearly shown with their sub structures in the figure. It also shows all the linkages that DSA identifies amongst the substructures in the benchmark.

Both the data structures identified were global in nature and were recognised as global by the DSA, as shown by the presence of G flags in the figure. There were no local data structures in the figure and rightly DSA did not identify any. This shows DSA's strength in performing escape analysis for the different data structures. None of the nodes in this benchmark got collapsed. This again showcases DSA's strength to infer type information, i.e. DSA is field sensitive. There were just single instances of the above mentioned data structures.

Metrics: N=2

N as mentioned earlier is the total no of SDSI found. In the mcf code there are are two global static data structure instances : one which corresponds to the main data structure network ant the other which acts like a basket containing the list of arcs. The figure also shows that the "arc" sub-structure is pointed

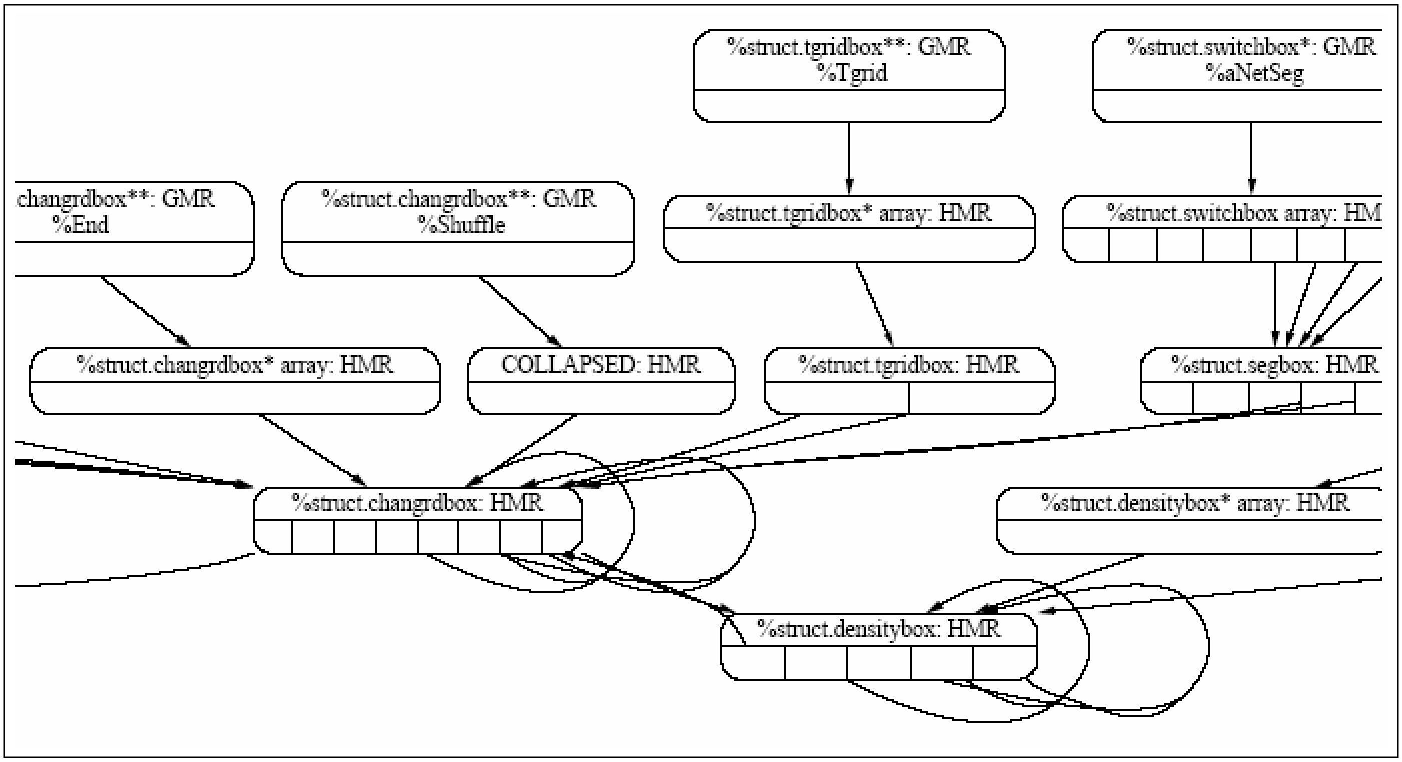

Figure 8: CBU graphs of doubly linked lists of changridbox and densitybox in twolf

to both by the node sub-structure and the network data structure. Though this example shows DSA's strength of being field-sensitive it also presents one of its weaknesses. Due to the "unification" property of the DSA two distinct instances of the "arc" structure (net → arcs and net → dummy-arcs) are showed as one node in the figure.

G=2

Both the SDSIs are correctly depicted as being global.

L=0

No local SDSis found.

K= 5

The network SDSI has 2 nodes whereas the basket SDSI has 3 nodes in it.

I=0

No SDSIs had greater than 1 disjoint instances in the same function.

O=0

No Collapsed nodes.

C= 2

There are two nodes which form a self cycle and hence are a part of two distinct collections. Both the node and arc structure are linked lists and hence are shown as self loops by the DSA.

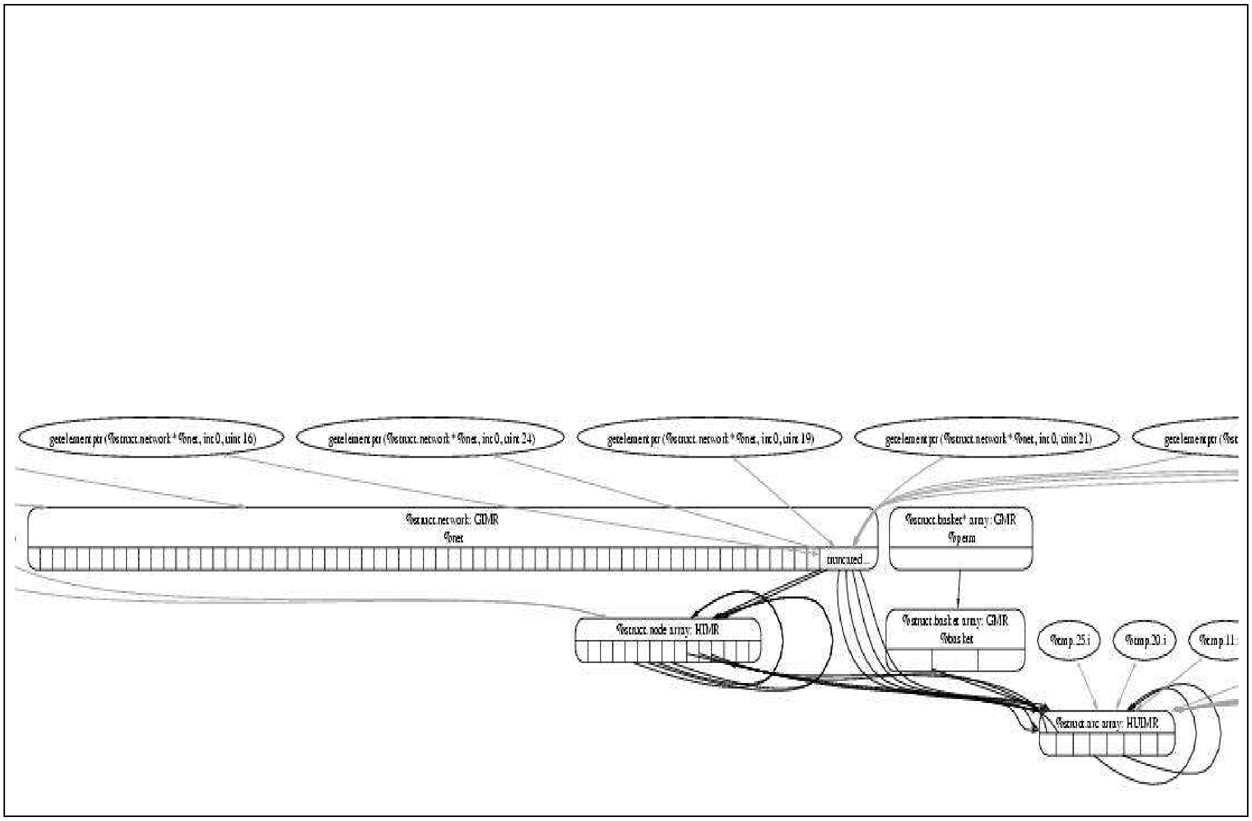

Figure 9: DSA Graph for the main function after complete BU analysis

8 Results for 197.parser-b

Summary:

197.parser is a part of SPEC CPU2000 benchmark. This link grammar parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. The code is written in ANSI C. 181.mcf has a raw LOC of 11391 with a total of 10086 memory instructions in LLVM representation. The parser benchmark used a custom memory allocator to handle its memory operations (malloc and free). These custom memory allocators were overridden so that malloc and free get used.

Data Structures:

Some of the important structures in this benchmark:

a). c_list_struct** : This structure as shown in Figure 10 is an array of pointers to array of c_list_struct. The c_list_struct is also clearly identified in the figure as being a linked list with a pointer to a connector_struct. Again the power that DSA has due to its inherent field sensitivity property helps us to identify the differnt pointer fields in the structure.

b). Disjunct Struct : This structure as shown in Figure 11 is a simple linked list with pointers to the Connector Struct. Again DSA's field sensitivity comes into picture.

c). Match_Node_struct ** : This structure as shown in Figure 12 is an array of pointers to array of Match_Node_struct. The Match_Node_struct is also clearly identified as a linked list, again due to field sensitive DSA algorithm.

d) Word_File_Struct : This is a Global structure and is clearly marked G in the Figure 11.

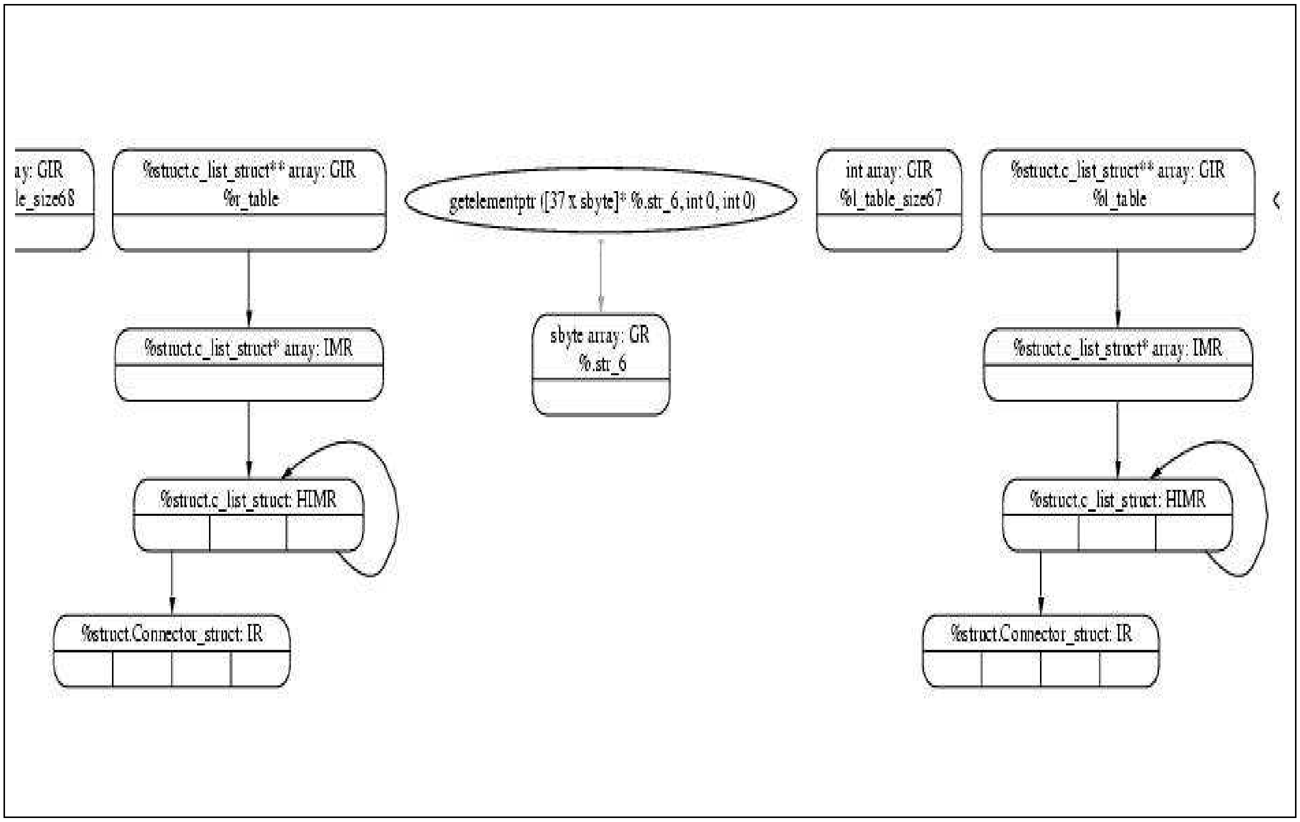

Figure 10: CBU graph for one of the functions

As shown in all figures above the DSA algorithm clearly identifies the local and the global data structures in the benchmark. This benchmark had a good number of both global and local data structures and DSA comes out true in all the cases identifying the correct scope of all the data structures. This strength of DSA is due to its strong escape analysis characteristic.

As shown in Figure 11 and Figure 13 the DSA correctly identifies the disjoint instances of the Connector struct. Also we see in Figure 12 disjoint instances of Match_Node structure. These all showcase one of many DSA's strength. The reason behind DSA identifying these disjoint structures is the context sensitivity of the DSA algorithm.

Figure 13 is a very good example demonstrating the power of DSA. Not only does it identify the correct structure hierarchy but it also identifies the distinct instances of the Connector substructure. The Disjunct structure and its substructure do not escape out of this function, build_sentence_disjuncts() and this is correctly depicted by the absence of the 'I' (Incomplete) flag in the nodes.

197.parser is a very good benchmark to showcase the actual power of DSA. It has many distinct data structures with varying lifetimes.

Metrics:

DSA has some good results for the 197.parser benchmark, specially because it has large number of data structures and most of its instances turn out to be disjoint.

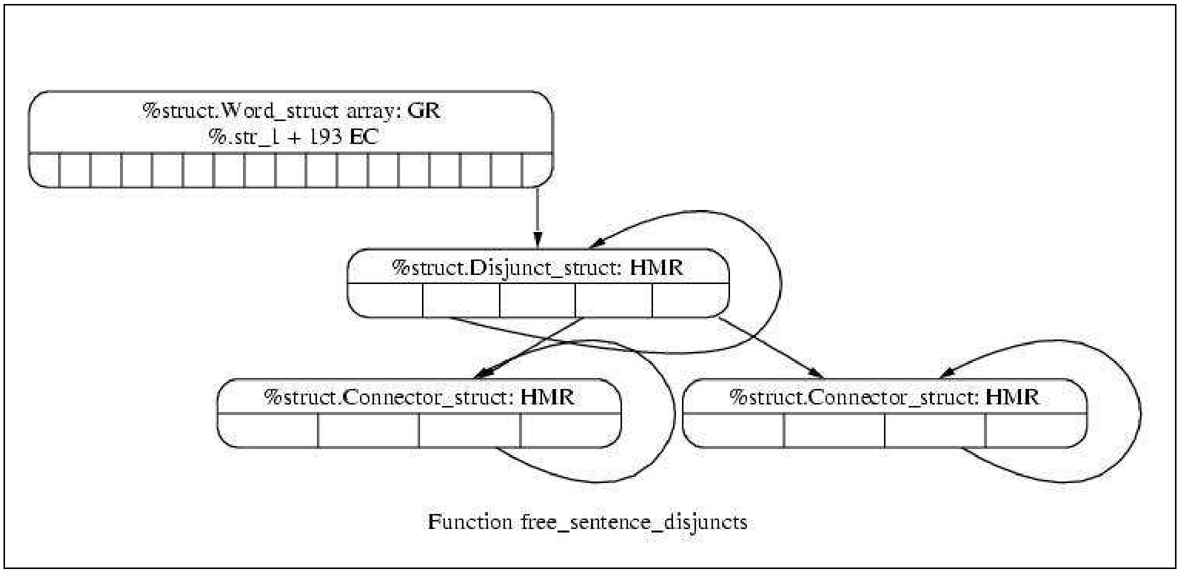

Figure 11: CBU graph for free sentence_disjuncts()

N=40

Large number of data structures are present in the code. Some of the data structures were mentioned above

G=20

All of the global SDSIs were allocated in the main() function.

L=20

Half of the SDSIs were local to the function were they were initialized. All of the data structures allocated outside main() are local and DSA shows them to be local. Examples of such kind have been discussed above.

K=105

197.parser had quite a large no of heap allocated nodes.

I= 53

As shown in Fig 5 , there were in total 53 disjoint data structures identified in the code. Some disjoint instances have been discussed above.

O= 3

Though there were all these good results, DSA could not identify the type of some nodes and finally there were also 3 collapsed nodes finally. See Figure 14.

C= 55

There a large no of nodes which form a completely connected structure or is a part of self cycle.

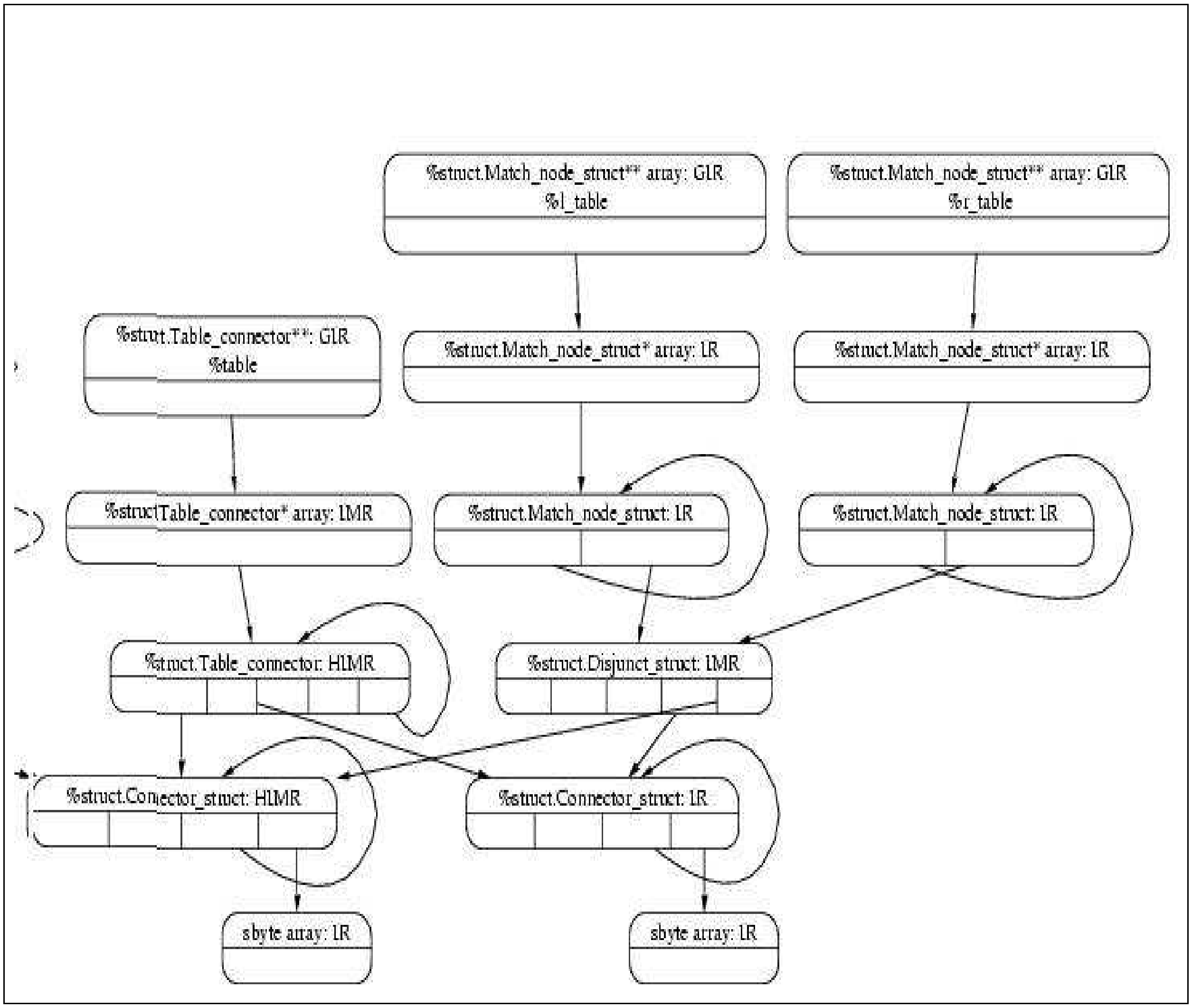

Figure 12: CBU graph for conjunction_prune()

9 Results for eon

9.1 Summary:

252.eon is a probabilistic ray tracer, and is part of SPEC Int 2000. A ray tracer is a program which renders three dimensional images by drawing rays from the "camera" and computing intersections to find the specific color to make each given pixel in the image. A probabilistic ray tracer is one in which ray direction is determined randomly. The implications of this, for this particular benchmark, is that there is less memory locality because vastly different areas of the scene graph (the structure which organizes the objects in the scene and provides a space partition to limit the number of computed intersections) may be accessed near each other temporally. For example in a normal ray tracer it would be common to draw rays into one particular division of the scene graph multiple times in succession, but with the random element there is a high likeliness of switching between divisions frequently. Eon consists of 23,653 lines of C++ code, not including comments or blank lines. There are 166 calls to new and 87 calls to delete.

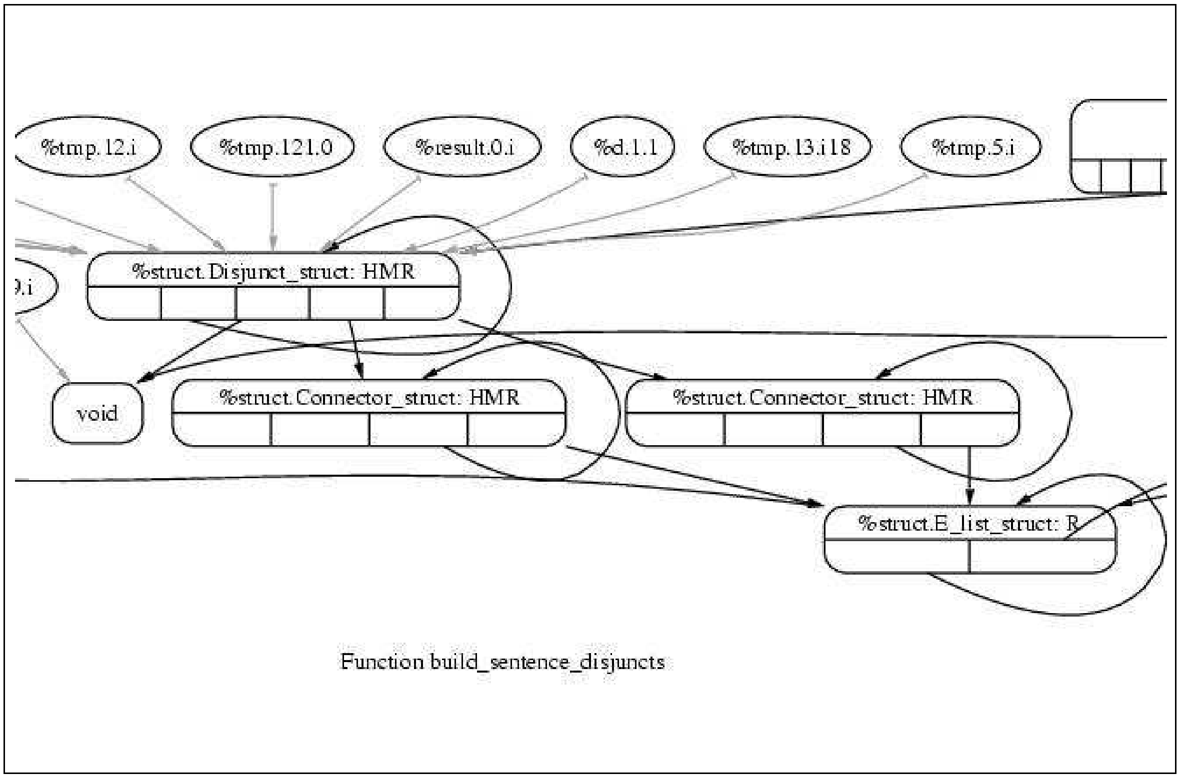

Figure 13: Distinct instances of the Connector_Struct substructure

9.2 Data Structures:

There is one major data structure in the benchmark, representing the scene graph, class mrScene, which is wrapped in an eonImageCalculator class. Figure 15 shows what the data structure graph looks like for an eonImageCalculator at an incomplete stage, before it is collapsed.

We can see in the above figure that there are several binary search trees (BSTNode) in the eonImageCalculator. The mrScene object is a field of eonImageCalculator and it is the source of all the edges connecting to the BSTNodes (where the figure says truncated). This data structure is not global per se (from the program's perspective), but has lifetime throughout the running of the program because it is created in main. All of the elements of the scene to be rendered are contained in the mrScene field of the eonImageCalculator.

9.3 Metrics:

N=1
The only true SDSI is eonImageCalculator, a field of which is a mrScene object.
G=1
eonImageCalculator is global because it is collapsed with global objects.
L=0
The eonImageCalculator is global.
K= 1
The eonImageCalculator SDSI has 1 node, because the individual nodes of the structure are collapsed

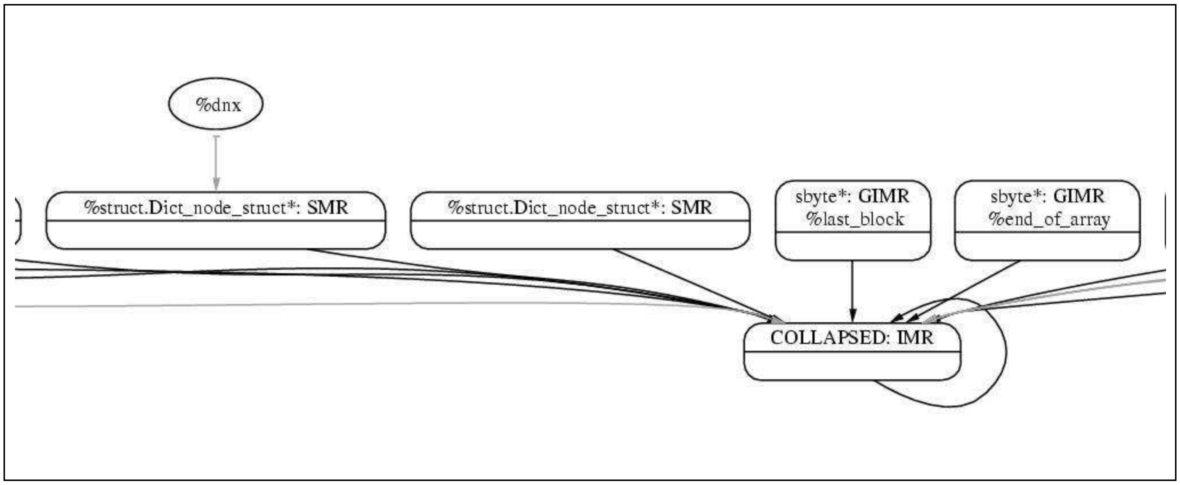

Figure 14: DSA Graph for function read_entry() which has a collapsed node

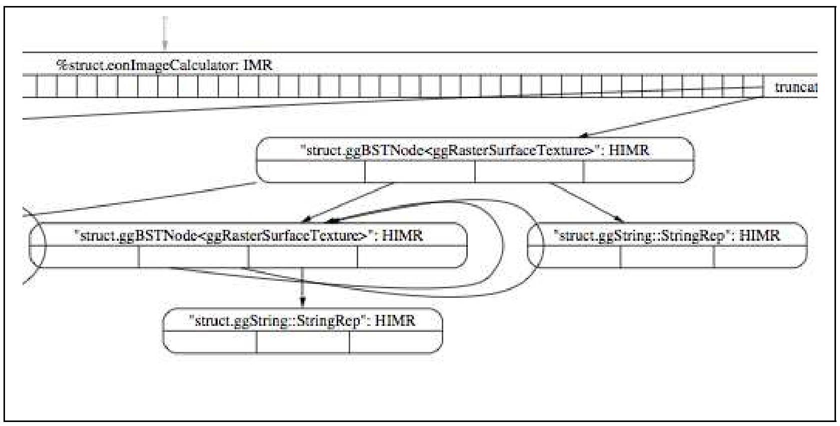

Figure 15: This is part of the graph for the eonImageCalculator constructor, unfortunately there is no graph marked complete for this data structure, it is collapsed in main. The BSTNode nodes form a recursive tree.

into one node in main.

I=0

There are no disjoint SDSI's, because there are no disjoint instances of one static data structure in the program, with the same type. There are a few instances of trees that are disjoint, but they are part of mrScene, and are templated to different types.

O=1 eonImageCalculator is, unfortunately, collapsed in main.

C= 1

Several collections point to the collapsed node.

10 Conclusions

Briefly, we summarize the conclusions of our study as follows. Please refer to Table 1 for the values of metrics referred to below:

1. In all but one (`eon`) of the six programs, DSA successfully distinguished the important *kinds* of

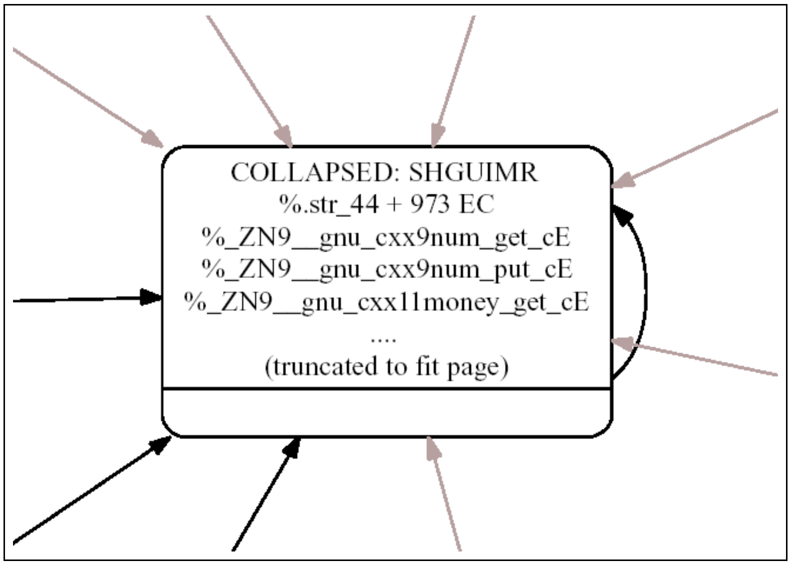

Figure 16: This is the collapsed node from main that represents the eonImageCalculator. It actually has around 200 nodes pointing to it, it has been pared down to fit the page.

	Static #LDS			Individual Nodes			
Benchmark	**N**	**G**	**L**	**K**	**I**	**O**	**C**
181.mcf	2	2	0	5	0	0	2
197.parser	40	20	20	105	53	3	55
300.twolf	23	21	2	97	6	7	29
espresso	33	7	26	148	67	0	47
fpgrowth	3	0	3	9	0	0	2
eon	1	1	0	1	0	1	1

Table 1: Observed statistics for linked data structures

data structures we have manually identified as non-trivial, logically distinct data structures in the codes (column N shows the total number of such data structures). For example, we identified 23 such data structures in `twolf`. In `eon`, however, a complex data structure with many sub-structures became merged with unrelated global arrays (and became collapsed).

2. In a number of cases (column I), DSA identified 2 or more disjoint instances of nodes representing recursive data structures, requiring a context-sensitive analysis.

3. In a number of other cases (column C), however, a DS node has multiple instances (e.g., a list of lists) and DSA is unable to distinguish these instances. A powerful, flow-sensitive analysis would have been required to distinguish most of these cases.

4. Most DS nodes of these data structures (K-O) have accurate type information. The exceptions are usually minor, except for the collapsed node in eon.

5. DSA accurately identified the lifetimes (local, L, or global, G) of all the data structures we have examined so far, except an important case in `fpgrowth`, discussed next.

6. In `fpgrowth`, a recursive function creates a new tree and a new "item table", plus three other new objects on each recursive call. DSA was unable to prove the new tree and item table are local to

the function (because they are passed to the next recursive call) but is able to do so for the other three objects (which are not). The recursive polymorphism of Chin et al. [1] may confine the local tree as well.

Overall, on the positive side, our inspection has shown that DSA is successful at distinguishing different kinds of data structures, their type and lifetimes, and in many cases, is successful at distinguishing distinct instances of such structures via context-sensitivity. We found only two significant cases (in `mcf` and `twolf`) where unification caused unrelated data structures to be merged. We consider this a positive result because unification is a crucial factor in achieving a very fast analysis. We believe this result is achieved because of our focus on heap-allocated data structures (which are difficult to track precisely with or without unification) and because context-sensitivity eliminates some of the key weaknesses of unification (for function parameters).

On the other hand, DSA is unable to distinguish instances within a collection (requiring flow-sensitivity) or instances requiring context-sensitivity within a recursive computation. We believe lack of flow-sensitivity is by far the greatest limitation, but also would be a particularly expensive feature to add to an inter-procedural algorithm like DSA.

References

[1] W.-N. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *PLDI*, Washington, DC, June 2004.

[2] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, pages 310–323, New York, NY, USA, 2005.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 1–12, 2000.

[4] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Comp. Sci. Dept., Univ. of Illinois, Urbana, IL, May 2005.

[5] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[6] C. Lattner and V. Adve. Data structure analysis: A fast, flow-insensitive algorithm for analyzing linked data structures. Submitted for publication, Nov. 2005.

[7] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1), Jan. 1998.