



McGill University  
School of Computer Science  
Sable Research Group



---

## Dynamic Shape and Data Structure Analysis in Java

Sable Technical Report No. 2005-3

Sokhom Pheng and Clark Verbrugge  
{spheng, clump}@cs.mcgill.ca

October 27, 2005

---

[www.sable.mcgill.ca](http://www.sable.mcgill.ca)

## Abstract

Analysis of dynamic data structure usage is useful for both program understanding and for improving the accuracy of other program analyses. Static shape analysis techniques, however, suffer from reduced accuracy in complex situations. We have designed and implemented a dynamic shape analysis system that allows one to examine and analyze how Java programs build and modify data structures. Using a complete execution trace from a profiled run of the program, we build an internal representation that mirrors the evolving runtime data structures. The resulting series of representations can then be analyzed and visualized, and we show how to use our approach to help understand how programs use data structures, the precise effect of garbage collection, and to establish limits on static techniques for shape analysis. A deep understanding of dynamic data structures is particularly important for modern, object-oriented languages that make extensive use of heap-based data structures.

## 1 Introduction

Data structure or *shape analysis* techniques summarize dynamic data connectivity, with the goal of improving alias analysis [9], automatic parallelization [11], optimizing garbage collection [21], debugging, or as part of a general understanding of program behaviour. Investigation of data structures shape and usage is particularly important for programs which make extensive use of heap data, such as in Java and other object-oriented languages. Static approaches to shape analysis, however, suffer from overly-conservative approximations, easily induced by temporary data structure inconsistencies during updates and modifications.

In this paper we investigate shape analysis from the perspective of dynamic analysis. Using complete traces of Java program executions, we reconstruct the entire history of heap-based data as it is changed through program modifications. This allows for the construction of data structure snapshots and animations, visually illustrating evolution of program data, and also encoding a variety of properties of interest, including overall shape, age of data, node types, reachability, and so on. For large benchmarks, data sizes can become impractical, and so we also provide a more abstract and scalable numerical approach that summarizes the distribution of shape over time. We demonstrate our techniques on a selection of small and moderately large Java programs.

Data we dynamically gather provides lower limits on the potential accuracy of static, conservative shape analyses. Specific and interesting program behaviours and programming styles are also easily discernible through our visualization techniques. We are, for instance, able to show and measure the extent of and variation in garbage data carried through program execution (*GC drag* [20]). Similar strategies can be applied to represent and understand arbitrary other data structure properties.

### 1.1 Contributions

Specific contributions of our work include:

- The design and implementation of a framework for capturing the complete dynamic evolution of data structures in Java programs.
- We present two new techniques for data structure visualization: a series of snapshots that can encode current and historical data structure properties, and a numerical, summary assessment that can represent arbitrarily large data sets.

- We give experimental results on the data structure usage of a number of benchmark programs, including non-trivial programs in the SPEC JVM98 [22] and JOlden [3] suites.

In the next section we discuss background and related work on shape and dynamic analysis. In Section 3 we describe the general design of our shape analyzer and the associated research challenges. Section 4 then gives analyses results performed on a set of tiny, small and reasonably large benchmarks. Finally, we discuss future work and conclude in Section 5.

## 2 Related Work

Our approach combines two main techniques, dynamic analysis and shape analysis. These have historically been relatively orthogonal pursuits, and so we discuss them separately below.

### 2.1 Shape Analysis

Shape analysis techniques vary from implementing a whole new language for identifying data structures to summarizing them using specialized graphs.

A frequent, and early approach to identifying data structures is to allow the programmer to provide high-level information through program annotations. Hummel et al., for instance, define static annotations to data structures in order to help the compiler identify opportunities for parallelizing transformations [12]. A similar approach is described by Fradet and Le Métayer, who define a new language annotation that integrates the notion of shapes into the C language [7].

Many have tried identifying data structure shape without modifying the source code. Ghiya and Hendren show how the conceptually simple categorization of data structures into *tree*, *DAG*, or *cycle* can be sufficient for compiler optimization [9]. More detailed approaches attempt to model data using various kinds of graph abstractions. Klarlund and Schwartzbach's *graph types* build a representation as a grammar describing data structures having a backbone, such as doubly-linked lists [14]. Wilhelm et al. [23] define *shape graphs* to represent structural properties of data structures. Corbera et al. combines static shape graphs with abstract storage graphs to give a more precise shape analysis [4]; their techniques were improved by Navarro et al. by approximating the data structures in a graph combining memory locations having similar patterns [16]. Recently, Hackett and Rugina described a way of breaking down the entire shape abstraction into smaller component and analyzing them separately [10].

While most work done on shape analysis has been done statically on C code, Bogda and Singh have done some exploratory work on shape analysis for Java code at run-time [1]; good results are possible, but mainly under repeated execution scenarios.

Our own work here also includes aspects of data structure visualization. Visually representing the heap is an existing concern for debuggers [24], heap analysis tools [18], and visualizing profilers in general [19]. Most shape analysis studies, however, concentrate on the analysis more than depicting the analysis results, although there is recent work on defining structural shape properties suitable for visualization [13].

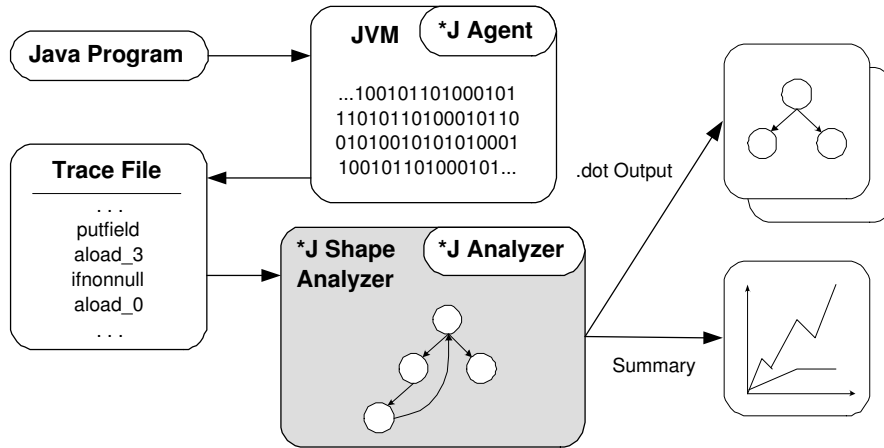


Figure 1: Design overview.

## 2.2 Dynamic Analysis

Dynamic program analysis can be performed online, or offline through the analysis of program execution trace files. Given the large resource demands of our precise shape analysis we have focused on the latter technique; inroads have been made to the former [1], however.

Trace extraction from Java programs often relies on the use of the Java’s built-in Virtual Machine Profiling Interface (JVMPPI), or its new replacement JVMTI (Tracing Interface). Brown et al. describe a framework, STEP, for profiler developers to encode general program trace data in a flexible and compact format [2]. JVMTI is also used by Dufour et al. in the implementation of \*J [5], a tool for dynamic analysis of Java programs used to generate Java program metrics [6]. Our work here builds on the \*J framework.

The Daikon project from MIT [17] and the Dynamo project from Indiana University [15] both provide online forms of dynamic analysis, differing mostly in usage. Both projects are based on observing runtime values and invariants to perform diverse analyses and optimizations. The Daikon project uses the information to report properties that were true over the observed period, which can then be used for testing and verification for example. Dynamo is a compiler architecture that uses the information to do runtime optimizations. The challenges of efficient online dynamic analysis are quite different from our exhaustive approach to trace analysis, but the invariant-based approach may be a useful basis for determining specific data structure properties.

## 3 Design

We begin with an overview of our design, followed by a description of the properties we can represent in the output, and the analyses that we can perform. Fully accurate dynamic data structure analysis implies significant research challenges in handling and representing large amounts of information; we describe the major problems and some solutions in Section 3.4.

The overall flow for our shape analysis system is shown in Figure 1. The first part of the process is data gathering. Java programs are executed in the JVM (Java Virtual Machine), and an attached \*J agent produces execution trace files of the running program. Trace files are then fed into the \*J shape analyzer. Here the input event trace is used to reconstruct the program data structures and their evolution over time. The \*J

shape analyzer may apply various analyses such as tree/DAG/cycle analysis, topological shape analysis, etc. The last part of the process is the output representation of the analysis data. Results can be communicated as literal snapshot or animated representations of graph structures, or in the case of larger outputs as numerical summary graphs.

### 3.1 \*J Shape Analyzer

The \*J shape analyzer relies heavily on \*J, which is a tool for dynamic analysis of Java programs [6, 5] and it comes in two parts. The first part is the \*J agent, which produces trace files containing events obtained from the JVM through the JVMPI (Java Virtual Machine Profiling Interface). The second part is the \*J analyzer, a framework for reading trace files and performing different analyses on that data. The \*J shape analyzer extends the basic analysis facilities of \*J.

For a complete and accurate analysis of runtime data structures we need complete data on heap objects and references, and all values which may be stored in reference fields. \*J provides both a complete trace of all instructions executed, and unique identifiers for all objects. We are thus able to reconstruct heap connectivity by tracking which object identifiers are subject and target of reference field writes; this includes reference arrays. The \*J shape analyzer reads events from the generated trace file and processes them one by one. For each event processed a corresponding update is applied to an internal structure that mirrors the program's heap nodes and their connectivity. This includes the removal of nodes due to GC. At each of these modification points, analyses are then run to determine the evolving properties of the data structure.

### 3.2 Data Structure Properties

From the mirrored representation of the program data structures we are able to find and show a variety of interesting and useful properties. Certainly type, or other node information can be easily included in any graphical representation. We can further encode complex, historical node properties such as relative age of its component nodes, and the data structure can also be examined more abstractly, e.g., in terms of reachability.

Node type in our representations is shown textually. However, since we are most interested in application objects, we distinguish application from library objects through colour as well, and this strategy can obviously be extended to many node properties. Figure 2 shows an example of this division, as well as a visualization of the *aging* property: as an object ages, meaning that it lives longer within the program, its colour becomes darker (in figure 2 this is applied only to application objects, not library objects). Observing age and type can be a useful way of understanding how a structure is constructed; in figure 2, for example, it is evident that the data structure is mostly built bottom-up, with application nodes near the tree root younger than nodes deeper in the structure.

Reachability in our system is easily determined. By tracking all object references we also know the set of all root objects, or entry points to the structure. Root objects include static variables, live local variables, and live method parameters. Thus by comparing the transitive closure of references with the set of all allocated but currently uncollected objects we can determine the set of dead objects, not reachable from the root set. This information can be visualized, showing the exact amount and (remaining) connectivity of dead, garbage objects the heap contains. Figure 3 shows a visualization of a data structure containing garbage data. Dead objects are drawn with dotted lines, and we can easily see how many there are and exactly how they are connected to each other and to the rest of the structure. Understanding how much data is carried in this way can be useful for garbage collector optimization [20].

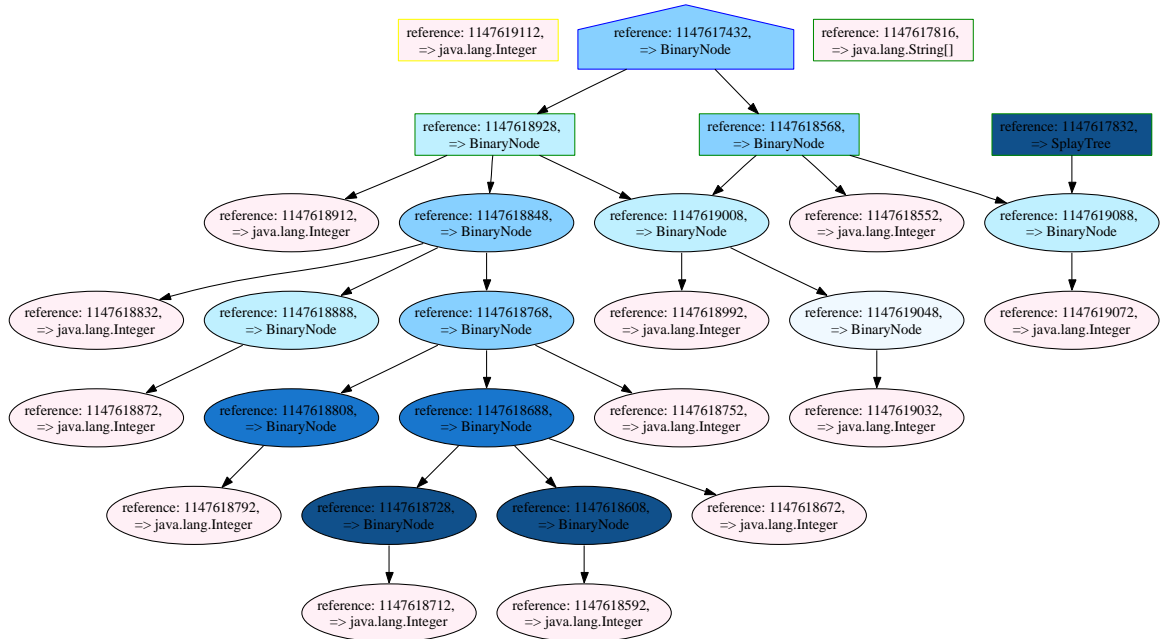


Figure 2: A data structure showing the aging property. Nodes are coloured according to their age (and type); all leaf nodes here are library objects, and all internal nodes application objects.

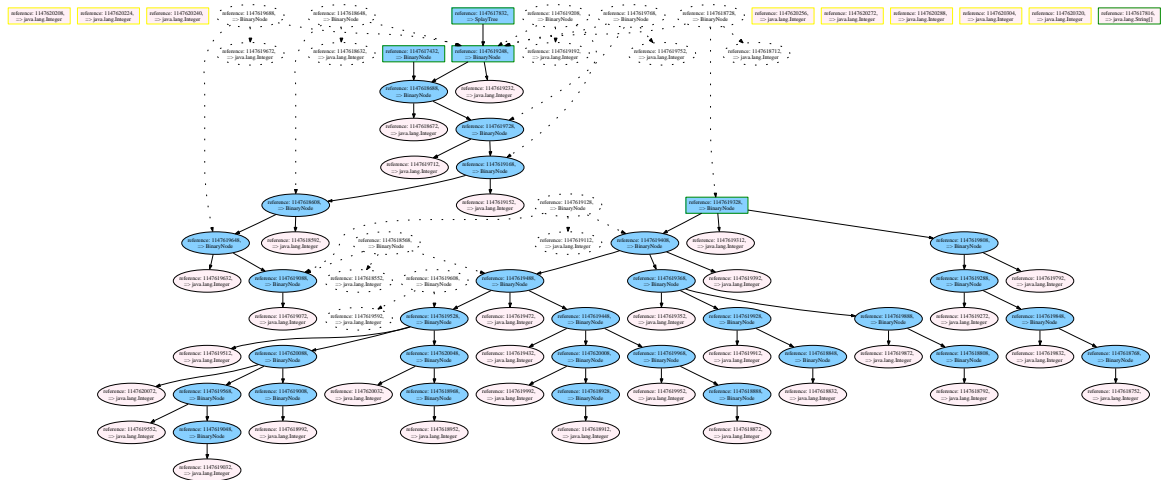


Figure 3: Showing garbage nodes in the data structure. Here unreachable nodes are drawn in dotted lines.

### 3.3 Analyses

The \*J shape analyzer has all necessary information to support the implementation of various analyses, including different summary and shape graph approaches, topological shape analysis, etc. We have implemented a basic tree/DAG/cycle analysis as a proof of principle, and also to investigate the quality and utility of this simple categorization.

Dynamically, a tree/DAG/cycle categorization is quite trivial to compute. From each entry point we simply do a depth-first search to determine whether the nodes reachable from that entry point represent a tree, a DAG or a cyclic graph. This information is then encoded in the graphical output; if the reachable nodes form a tree then the entry point is drawn as a rectangle, if the structure is a DAG then the entry point is drawn as a “house shape” (pentagon), and for cyclic structures a hexagon entry point is used. Although these qualities are usually associated with the data structure itself more than the entry points, it is also true that a structure may appear differently from different perspectives. Figure 2 shows examples of tree and DAG entry points into the same connected structure. By performing this analysis at each structure modification we obtain an evolving view of the data, at least in terms of tree/DAG/cycle composition.

### 3.4 Representation Concerns

The most obvious and direct representation of data structure evolution is as series of literal snapshots of the encoded data structures, as in figures 2 and 3. This is suitable for small tests, examinations of specific components, and for pedagogical pursuits, but unfortunately is not feasible as a general approach in most benchmarks. The large data sets that must be manipulated in the context of the analyzer impose strong constraints on the style of presentation, and also on the kind of data that can be gathered.

Tiny, test programs modify data structures only a relatively small number of times. More realistic programs, however, can perform a very large number of updates; the Jess benchmark from SPECjvm98, for instance, performs more than 48 million heap modifications. Examining all these snapshots is physically unrealistic for humans. Instead of generating snapshots for each modification we therefore only generate a snapshot every  $n$ th changes, for different  $n$  depending on the scale of investigation required. This can also help in reducing the computational cost of the analysis.

Snapshot animation itself is surprisingly difficult, even with external tools. In order to have a nice animation of the snapshots, we need to be able to incrementally add/subtract nodes and edges to an existing drawing while ensuring existing nodes and edges do not move. This preserves the location of nodes between snapshots, making node identity trivially obvious as frames change. Current open source and commercial tools for graph layout, however, focus on optimal, static representations, and do not in general attempt to locate nodes in the same place between drawings. This results in animation frames where graphs in successive frames may bear little visual relation to each other, and thus are not useful as a visual replay of data structure behaviour.

Improvements to drawing tools are possible of course. However, many programs also produce large data structures, whether or not they are modified frequently. Even a simple program such as BiSort from the JOlden benchmark suite generates more than 120 thousand objects—far too many objects for a drawing tool to handle, or to meaningfully show on a screen or in an animation. Interactive visualization techniques can improve this situation, but it is clear that animations, and even representative snapshots are simply not feasible in all situations. For the benchmarks we analyze in the subsequent section we have thus concentrated on alternative representations that draw only reduced, aggregate information on data structure properties, and not the data structures themselves.

Finally, we note that the amount of data that can be acquired through the JVMPI interface in \*J is limited. Early events in the virtual machine startup are not available (occurring before JVMPI is initialized), and data from native method executions is not reliably delivered. In our investigations we have restricted our analyses to application code, not startup in order to ensure we have a complete event trace.

## 4 Experiments

We have analyzed a number of benchmarks from the SPECjvm98 and Jolden benchmark suites. Below we describe the programs analyzed, and present examples of visualizations, both as snapshots and in terms of numerical summaries, along with discussion of interesting and relevant program features exposed by our analyses and visualizations.

### 4.1 Benchmarks

For space reasons we cannot show results for all the benchmarks we have analyzed, and instead show a selection of results from three basic categories. The first kind consist of tiny programs designed to test the framework, and also suitable for snapshot visualizations. We used two well-known algorithms, a splay tree implementation and a red-black tree implementation. Both programs construct a small tree and then delete some nodes; below we only present the SplayTree benchmark program.

More realistic, but still manageably small results are obtained by analyzing benchmarks from the Jolden suite. These are small but non-trivial programs that focus on use of dynamic data structures. Benchmarks shown here include Barnes-Hut, BiSort, Em3d, and TSP (Travelling Salesman Problem).

Our final category is of moderately large programs, taken from the SPECjvm98 suite. The benchmarks that are analyzed here are Jess and MpegAudio.

### 4.2 Snapshot Example

If a program is relatively small, and in general does not contain more than approximately 1k objects, a meaningful visualization of data structures updates can be produced where a snapshot is generated for each update. We use the *dot* tool in *GraphViz* [8] to layout the graphs, encoding node properties as discussed in sections 3.2 and 3.3.

In Figure 4 we show snapshots generated for a series of data structure updates performed in the SplayTree program. From a) to c) the splay tree has a node inserted, with an image for each modification: first the new node (and its associated data) are connected by pointing them to the node just below the root of the tree, and then the root pointer is redirected to the new node. Different graph layouts between snapshots make this progression less obvious, but this kind of visualization is still very rich in detail, and quite useful for understanding data structure operations and behaviour.

### 4.3 Analysis & Numerical Summary Results

Most of our other programs contain well more than 1k objects, and thus are not well-suited to the style of literal representation used for SplayTree. Instead, we have focused on the results of the tree/DAG/cycle categorization. For each of the benchmarks below we calculate the number of entry points that reach tree,

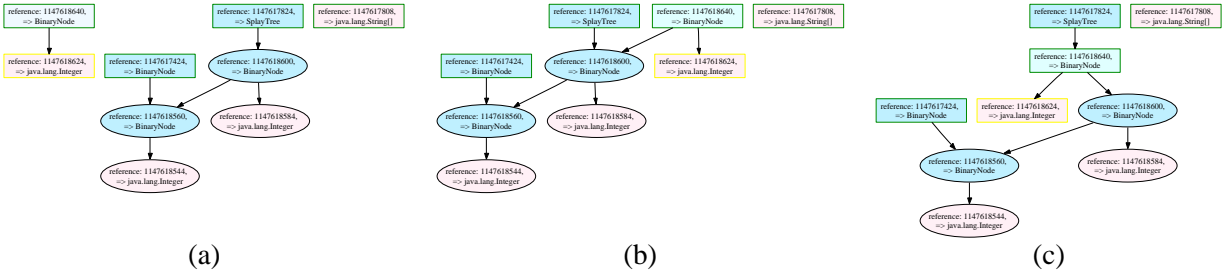


Figure 4: SplayTree snapshots. An existing pair of nodes (tree node and associated data) is inserted just below the root of the tree.

DAG, and cycle type data structures in the program, and plot this as it evolves over time (bytecodes executed). To keep data sizes and visual presentations manageable the data shown is actually sampled every 100–100k updates, as indicated in the individual descriptions.

The tree/DAG/cycle designation has one important limitation: single, unconnected nodes are considered trees. While this is true in a technical sense, many programs make extensive use of single node objects, and this obscures any understanding of more realistic tree usage. For this reason we actually show a 4-way categorization, separating single nodes into their own category.

To provide additional information we also show graphs of counts of live/dead objects over the same time axis. This makes it easy to see general trends in volume of data and garbage, and also allows for limited visual inspection of drag.

### 4.3.1 BiSort

BiSort performs two bitonic sorts, one forward and one backward. It works in two phases. The first phase is the tree construction, and the second phase is the sorting.

In Figure 5, we can easily see the first phase, where the tree is being constructed. A number of single nodes are allocated, and then consumed by construction of the base tree. At about 1/3 of the way through execution the program enters its second phase; here many changes are performed on the tree, and the number of tree structures becomes quite variable. As the tree is modified the data types fluctuate between DAG types and tree types in a complementary fashion: nodes are being rearranged, and not copied or deleted. Note that there are not in fact as many disjoint structures as the number of trees and DAGs would indicate; call chains and recursive calls in particular allow for the stack to contain multiple entry points to the same structure, magnifying the apparent number of structures.

A conservative static analysis on this program might be forced to conclude that the data structures overall are DAGs. Dynamically, however, the DAG stage is only intermediate, and trees dominate more than DAGs.

Figure 6 reinforces the observed phase behaviour of the data structures: objects are allocated (tree construction), followed by a long period of relative stability. Interestingly, there are no dead objects, an observation compatible with our claim that the data structure is modified by moving nodes, not adding or deleting.

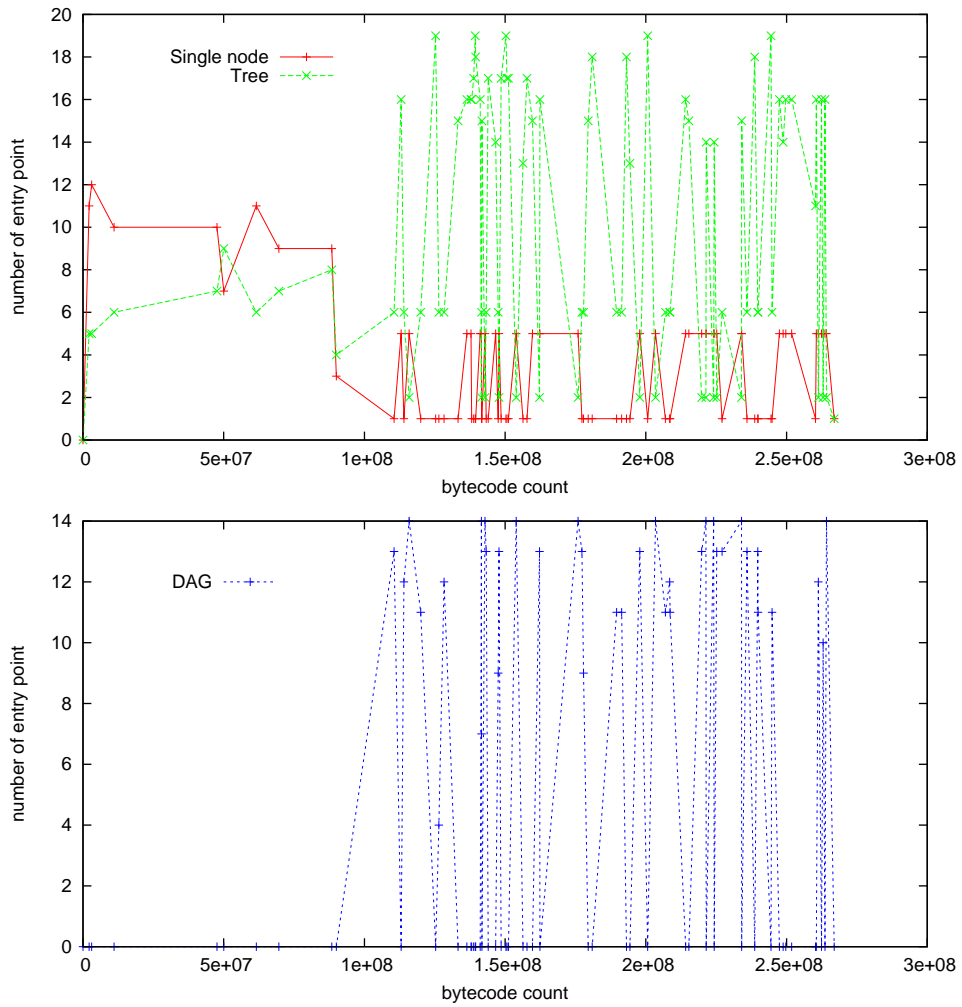


Figure 5: BiSort analysis results by bytecode for every 10k updates. The top figure shows single nodes and trees over bytecodes executed, and the bottom figure shows DAGs. There are no cycles in BiSort.

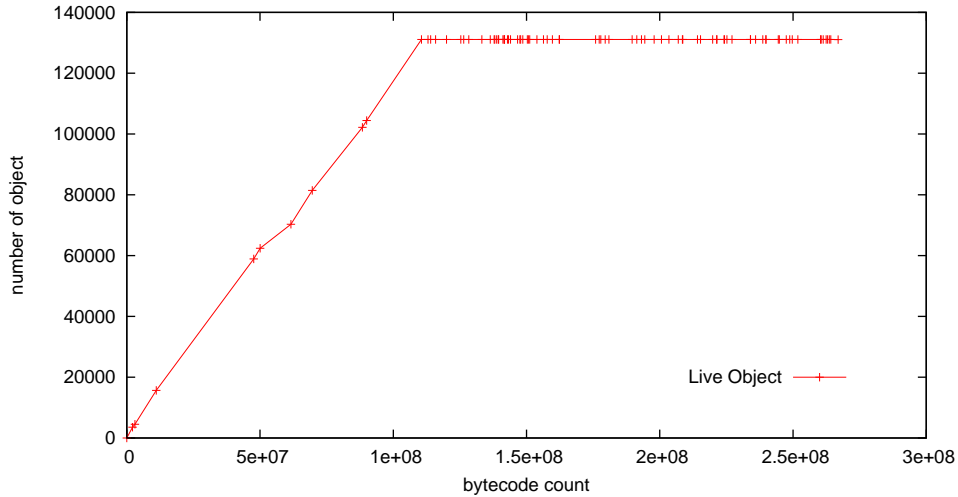


Figure 6: BiSort GC results by bytecode for every 10k updates, showing the number of live and dead objects over bytecodes executed. There are no dead objects in Bisort.

### 4.3.2 Barnes-Hut

Barnes-Hut solves the classic N-body gravitational attraction problem. Barnes-Hut works in two phases; this is not obvious by observing data structure type changes, but is clearly shown in the GC results graph of figure 8. The first phase is the tree construction, where a quad-tree is constructed, and the second phase is the force computation, where the tree is traversed.

From the graph in figure 7 it is evident that this program is quite dynamic in behaviour, and aggressive and frequent GC is used to limit the amount of accumulated garbage. As with BiSort there are no cyclic data structures at all. This is unsurprising for tree-based programs, but is also informative: it suggests, for instance, that the quadtree does not make use of parent pointers in child nodes.

### 4.3.3 Em3d

Em3d simulates the propagation of electro-magnetic waves through 3D object using nodes in an irregular bipartite graph to represent electric and magnetic field values.

In Figure 9, we can see that during the total execution of the program there are at most 5 trees and 1 dag at any point. Data structures in Em3D are quite few, and the ratio of live nodes to entry points suggests a limited number of larger data structures are used. In fact, there is mainly a table of linked lists.

Behaviour is relatively stable throughout this benchmark, at least until near the end of the program. At that point the data structures are reduced to a couple of single nodes and one tree. In this case we are able to see the effect of tearing down the data structures, something much less evident in the previous benchmarks. The conversion of data to garbage at the end of the program is confirmed by figure 10, where garbage rises as live objects reduce in number.

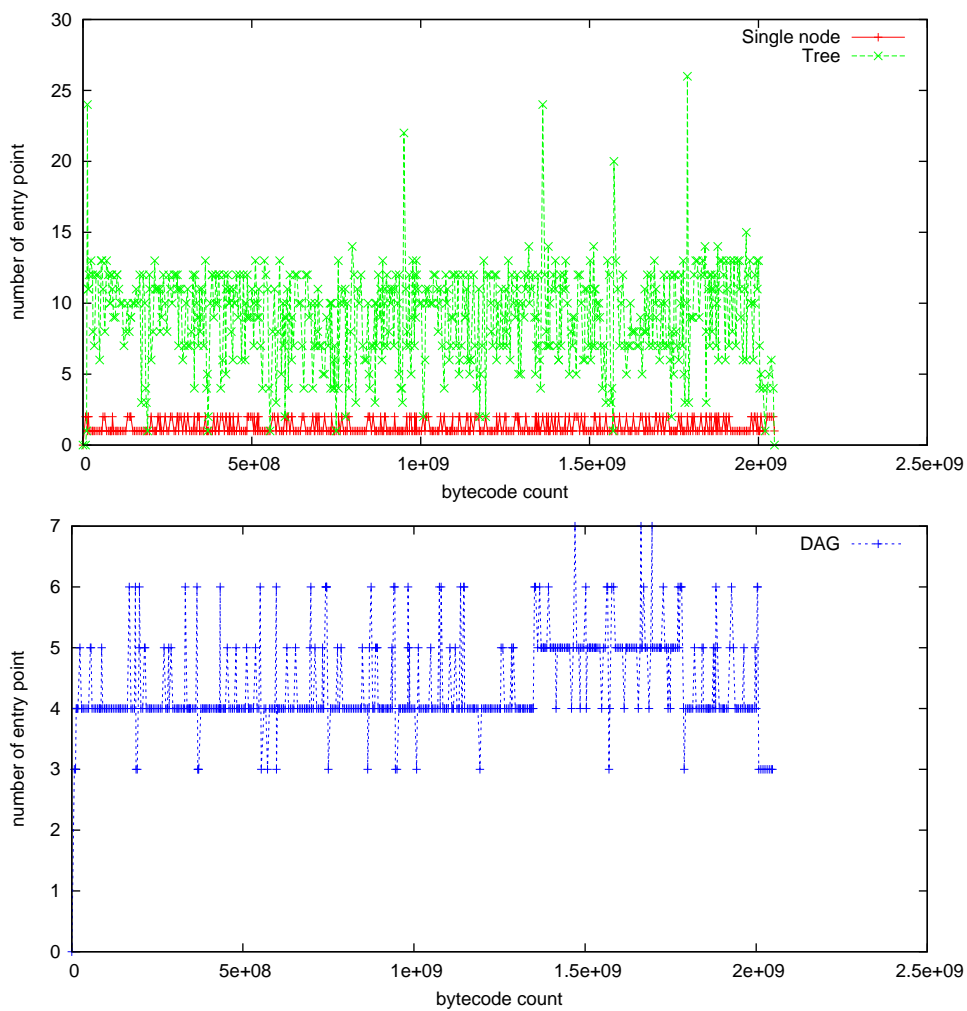


Figure 7: Barnes-Hut analysis results by bytecode for every 100k updates. On the top figure is shown the number of single node and tree entry points over “time” (bytecodes executed), and on the bottom the number of DAGs. Again, there are no cyclic structures.

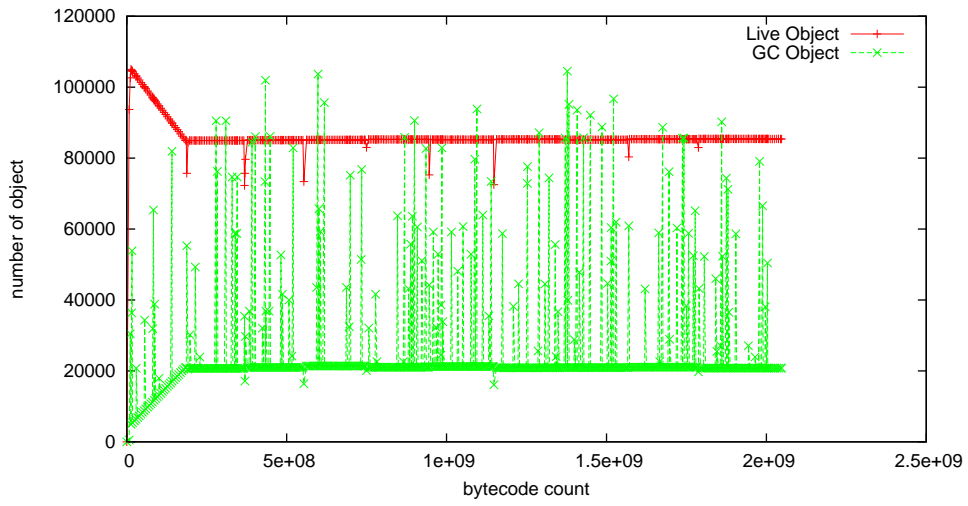


Figure 8: Barnes-Hut GC results by bytecode for every 100k updates, showing the number of live and dead objects over bytecodes executed.

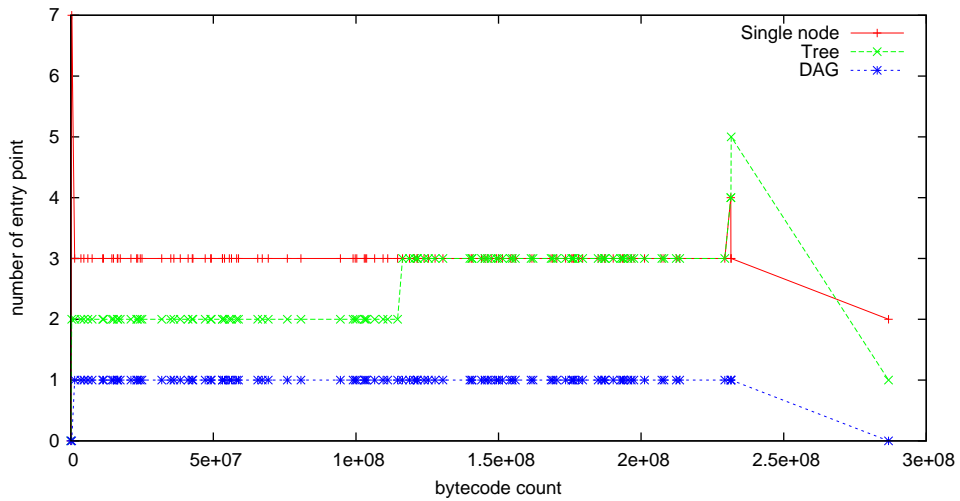


Figure 9: Em3d analysis result by bytecode for every 5k updates. Single nodes, trees, and DAGs are shown in this figure.

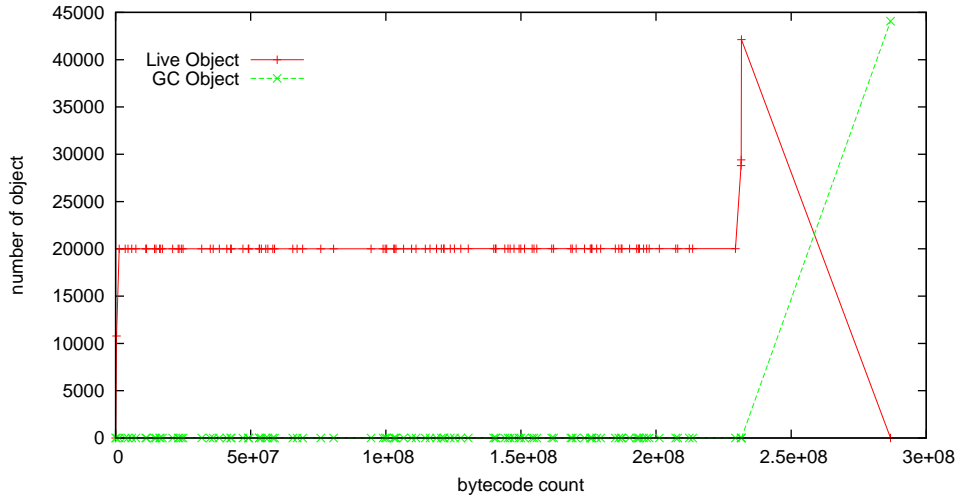


Figure 10: Em3d GC result by bytecode for every 5k updates.

#### 4.3.4 Power

Power solves the Power System Optimization Problem, where the price of each customer’s power consumption is set so that the economic efficiency of the whole community is maximized. It works in two phases. The first phase is the tree construction, and the second phase is the price computation.

Figure 11 shows there are only trees and single nodes present. This is consistent with the algorithm as it only construct a single huge tree.

From the bottom graphs of figure 11 and figure 12, we can see that the tree construction phase occurs within a very short time frame. However, from the top graphs, we can see that it consists of roughly half of the total data structure changes.

In Figure 12, we can see that within the computation phase, the number of live and dead objects remains relatively stable.

#### 4.3.5 Travelling Salesman Problem

TSP computes an estimate of the best Hamiltonian circuit for the Travelling Salesman Problem. There are two clear phases evident in both figures 13 and 14; a short initial phase constructing the problem, and a longer phase of analysis.

TSP is our first presented benchmark to actually include cyclic data structures. There are also a very large number of tree data structures, orders of magnitude more than single nodes, DAGS, or cycles. In fact the algorithm mainly builds trees, and the few cycles can be attributed to a double-linked threading of trees forming partial solutions to the input problem.

There is no garbage apparent in figure 14. However, the number of live objects decreases dramatically twice; there is necessarily some garbage generated by these reductions. In this benchmark the generation of dead nodes and their collection occurs between snapshots, leaving no direct evidence of dead nodes in our sampled results. Larger, more detailed graphs or actual numbers would reveal this difference. In terms of general trends, though, it is clear that TSP, particularly in comparison with Barnes-Hut, does not produce or

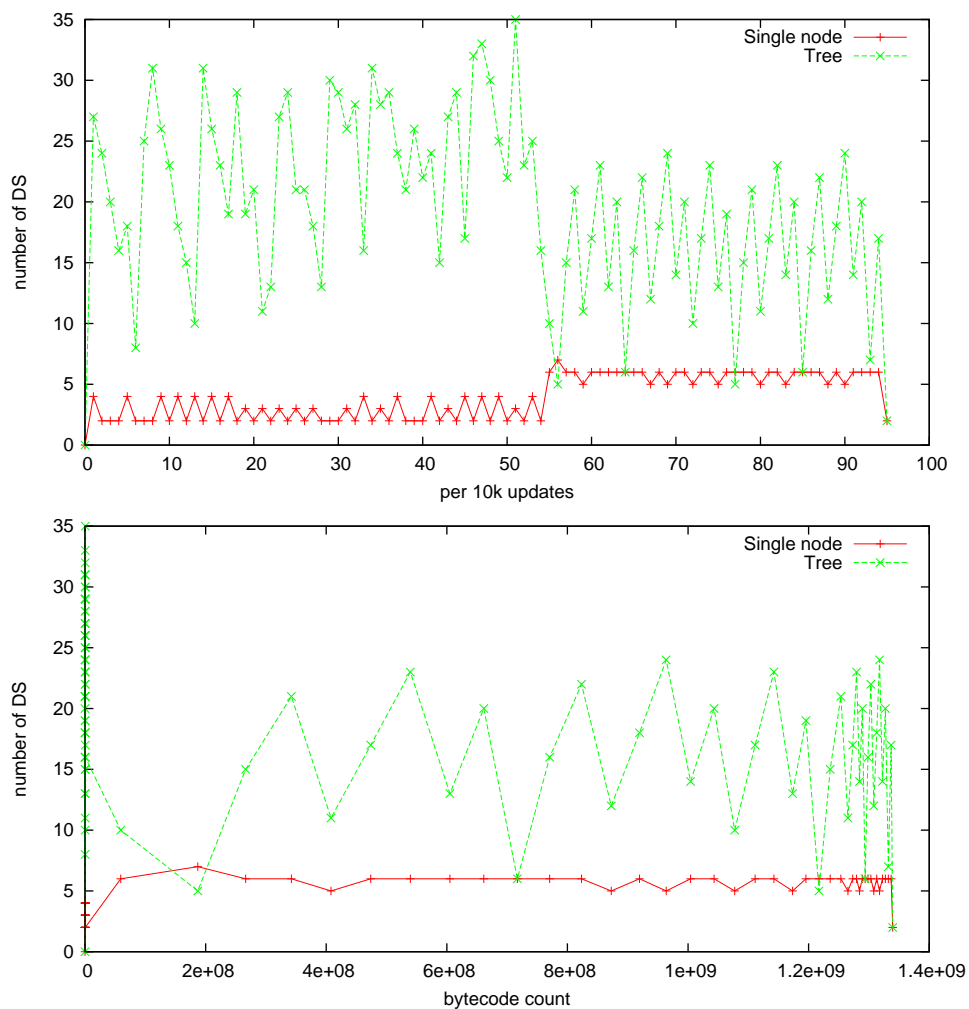


Figure 11: Power analysis result for every 1k updates. The top graph is plotted with respect to the total number of data structure changes, and the bottom graph with respect to the total bytecodes executed. Both graphs show the number of single nodes and trees.



Figure 12: Power GC result for every 1k updates. At the top the time axis is in terms of total data structure updates, and at the bottom in terms of bytecodes executed.

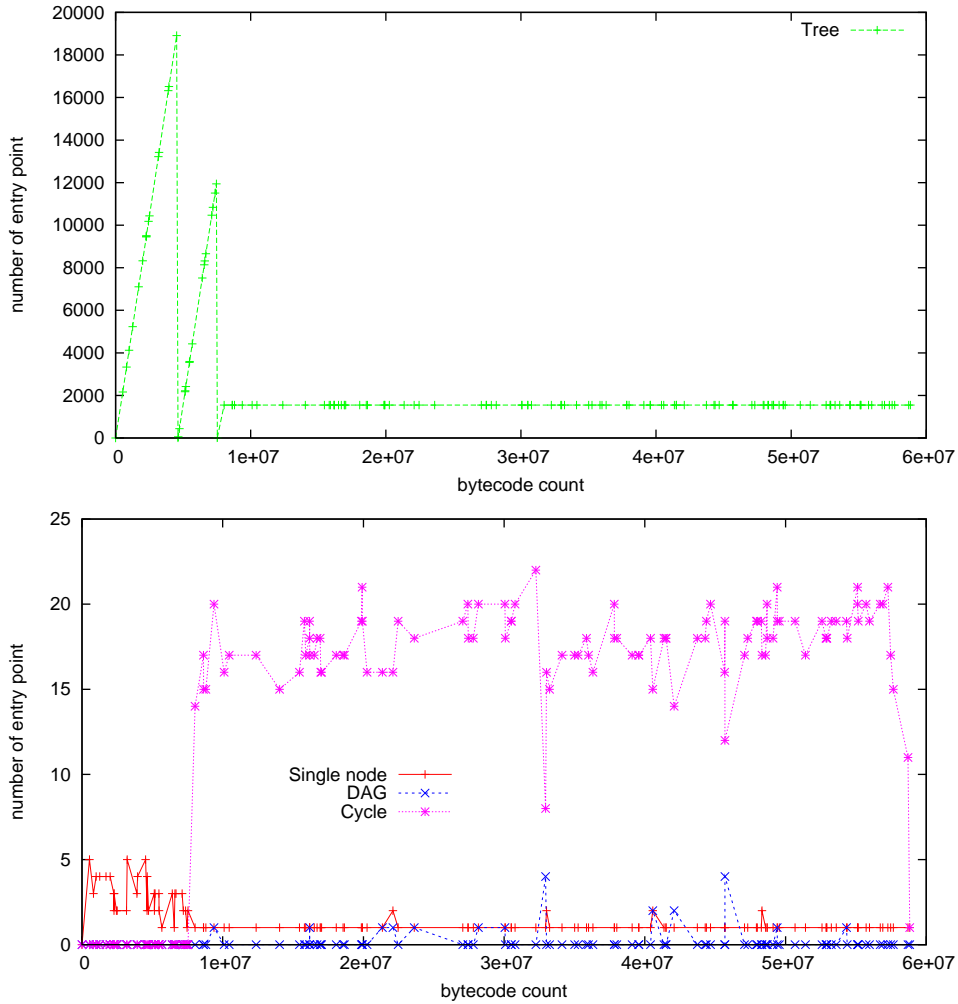


Figure 13: TSP analysis results by bytecode for every 1k updates. On the top are trees, and on the bottom single nodes, DAGs and cycles.

carry much garbage.

#### 4.3.6 Jess

Jess produces a lot of structures of all types, although most of them are single node objects, as shown in figure 15. There are no cycles, and there is a rhythmic pattern of tree/DAG construction. This behaviour roughly corresponds with the algorithm and input, which does repeated, tree-based searches to solve an input combinatorial problem.

Memory usage in Jess is more complicated than in the Jolden programs. From figure 16 we can see that a large number of objects are dead, usually many more than are live at any one time. Moreover, while the live set is overall stable, the number of dead nodes seems to have a general upward slant, increasing over time. This is also true of single node structures shown in figure 15.

We believe this to be an artifact of heap adaptation. Jess allocates a lot of temporary objects (single nodes). The heap pressure due to the use of temporary object allocations results in the heap being expanded to

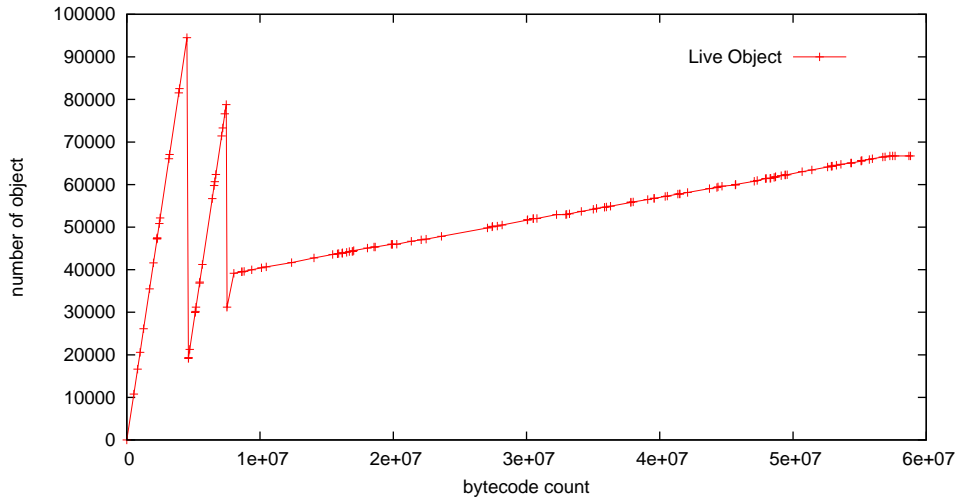


Figure 14: TSP GC results by bytecode for every 1k updates. Again, there are no dead objects evident in this graph.

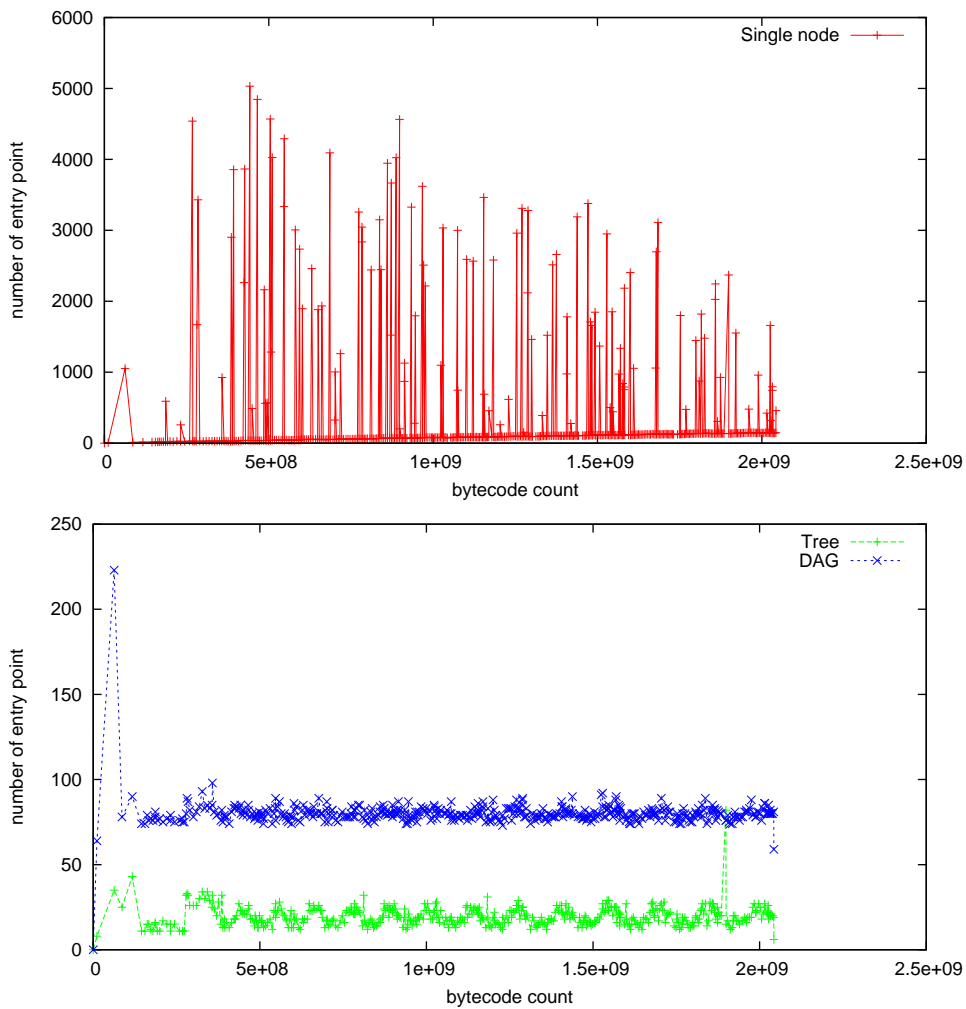


Figure 15: Jess analysis results by bytecode for every 100k updates. On the top are single nodes, and on the bottom trees and DAGS. There are no cycles.

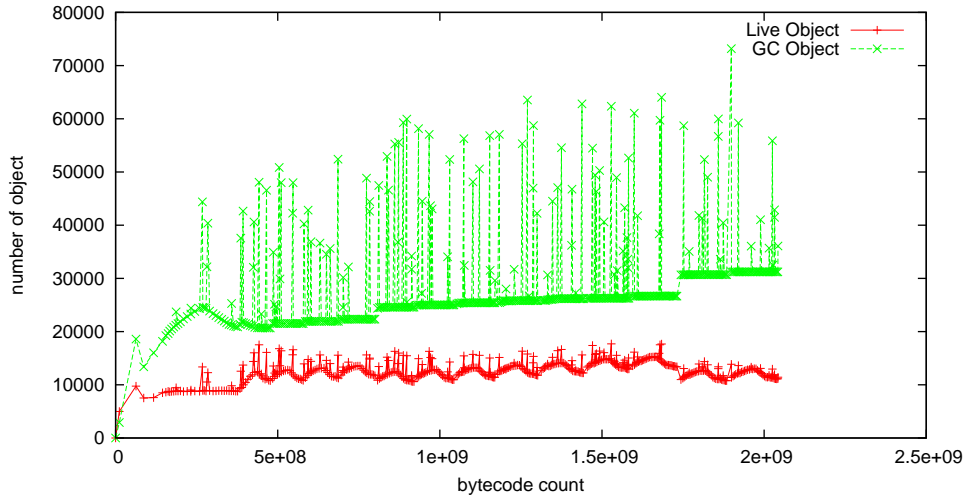


Figure 16: Jess gc results by bytecode for every 100k updates.

accommodate the perceived memory requirements. However, the core, necessary and retained data is not increasing, and a larger heap merely provides more room for garbage to accumulate. In this situation the amount of drag increases as the heap increases, suggesting that more aggressive GC rather than increasing heap size may result in more efficient execution.

#### 4.3.7 MpegAudio

Most of the benchmarks produce extremely similar graphs whether the time axis is formed of bytecode executions, or expressed in terms of data structure modifications: data structure updates are quite regular. MpegAudio shows this is not always the case. In the top graphs of figure 17 and figure 18, the number of data structures is shown plotted against total number of data structure updates. The data structure is smoothly constructed over the life of the program. The bottom graphs shows the same data plotted with respect to bytecodes executed. Here it becomes quite evident that the data structures constructed are built early on and used without significant dynamic changes for most of the program. The same behaviour is shown in the respective plottings of number of live and dead objects in figure 19.

### 4.4 Overall

For programs which make extensive use of heap structures a dynamic data structure analysis has the ability to provide a great deal of information about execution. Literal snapshots of data structures are most informative, but do not in general scale to being able to represent real program data. Even from a simple tree/DAG/cycle descriptions of data structures, however, a surprising amount of detail on program behaviour is discernible in our numerical summary graphs. We are easily able to see major phases in data structure usage and construction. The variation in data structure is also clear; few programs consistently and uniformly stick to one kind of structure, with most exhibiting fluctuations and transformations between at least trees and DAGs. This challenge to conservative static approaches may be compensated somewhat by the general lack of cycles, curiously appearing in just two of our six benchmarks. At least for these programs trees and DAGs are very much dominant.

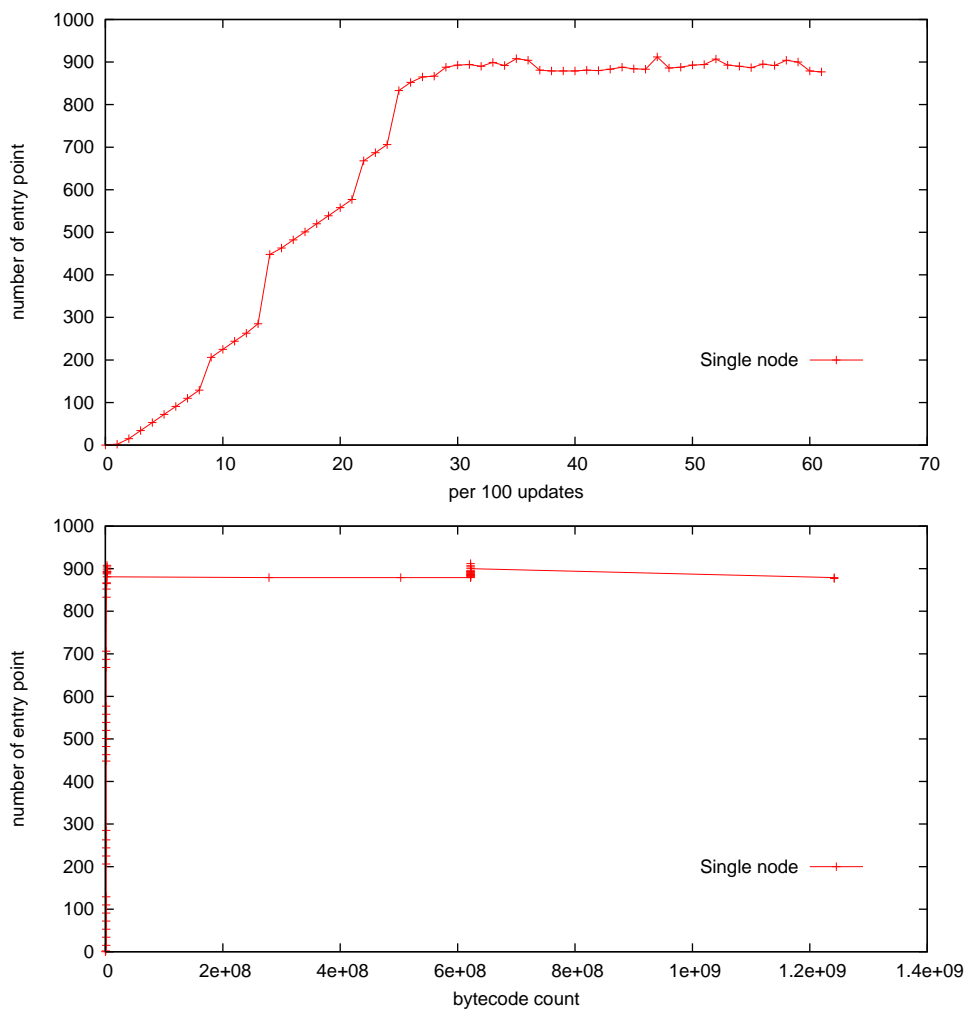


Figure 17: MpegAudio analysis result for every 100 updates. The top graph is plotted with respect to total number of data structure changes, and the bottom graph with respect to total bytecodes executed. These graphs show the number of single nodes.

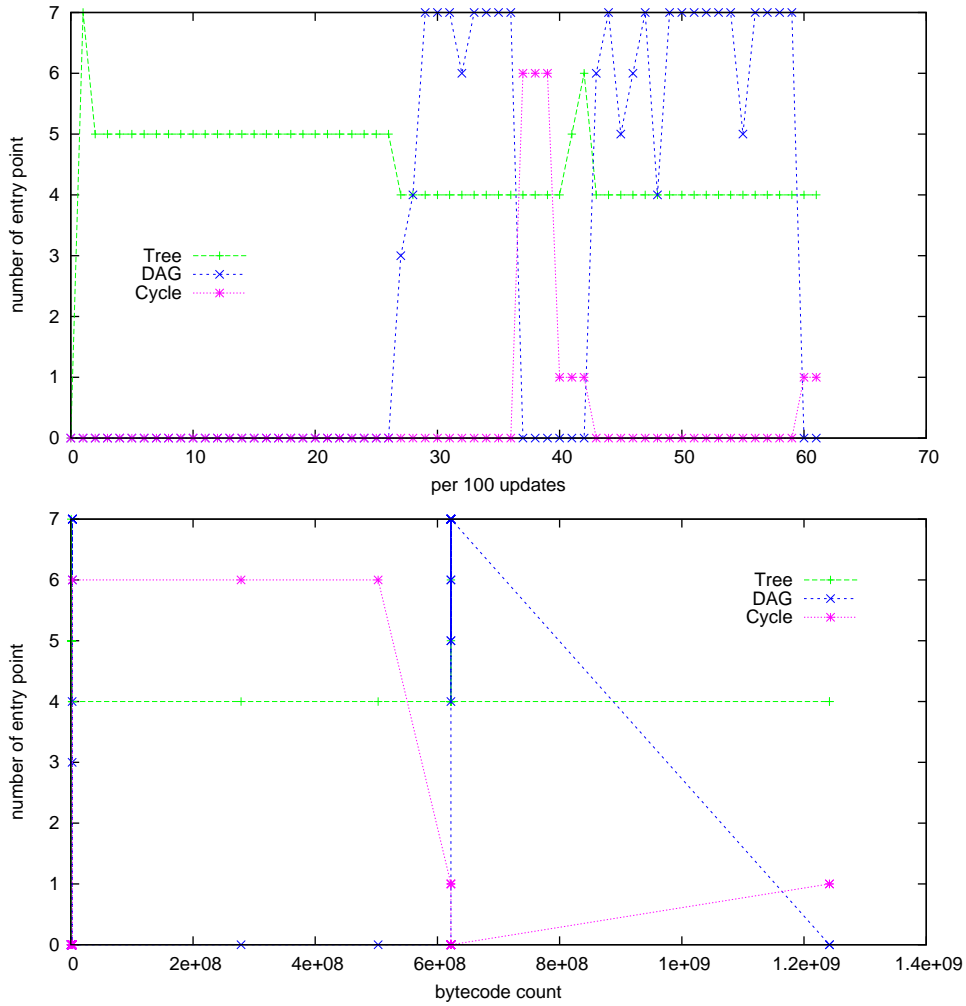


Figure 18: MpegAudio analysis result for every 100 updates. The top graph is plotted with respect to total number of data structure changes, and the bottom graph with respect to total bytecodes executed. These graphs show the number of trees, DAGs and cycles.

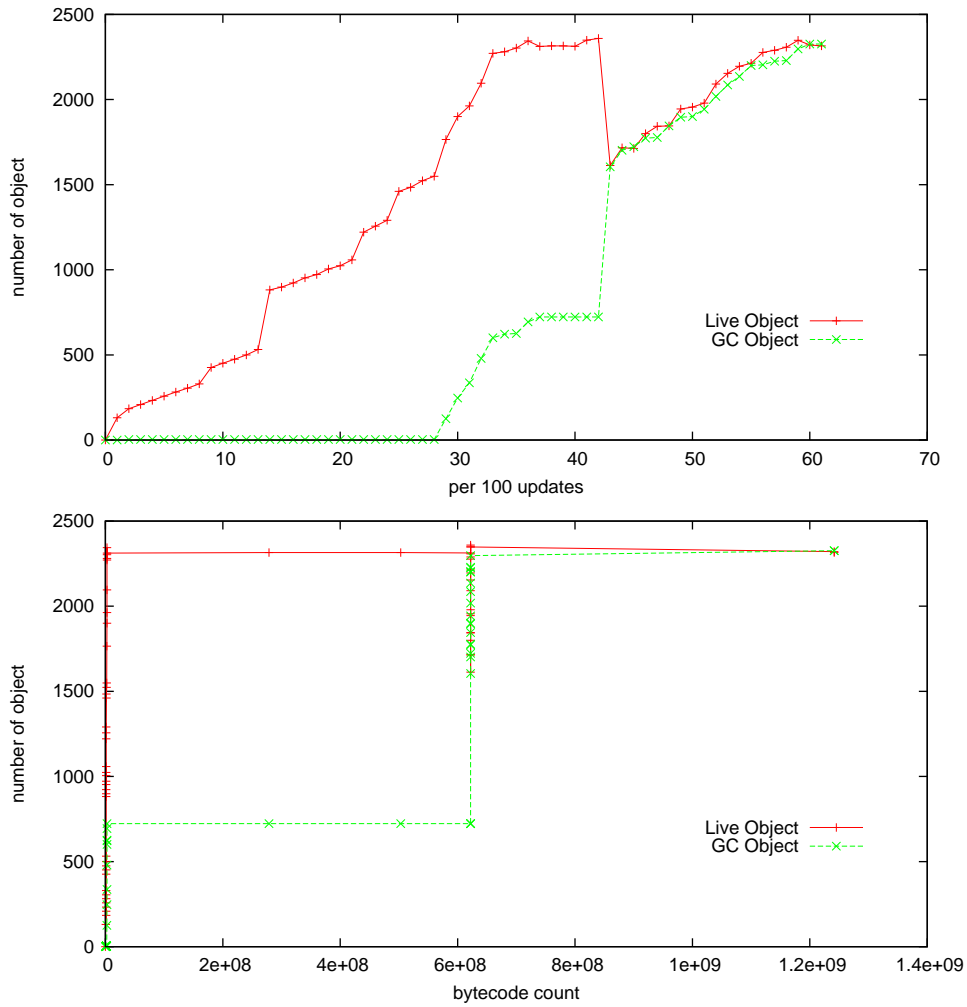


Figure 19: MpegAudio GC result for every 100 updates. At the top the time axis is in terms of total data structure updates, and at the bottom in terms of bytecodes executed.

The impact of garbage on memory use is also intriguingly variable. Barnes-Hut and Jess generate great amounts of garbage, and certainly in the latter case dragged dead objects can be seen as a potentially important factor. Other benchmarks, such as TSP and BiSort carry little to no garbage, and may benefit from a corresponding reduction in GC; these benchmarks are not strongly GC-dependent.

## 5 Future Work & Conclusions

Dynamic data structure analysis has the ability to show detailed information on various aspects of program behaviour. This can help identify program characteristics, heap usage, and provide general understanding of any calculable static or evolving dynamic data structure property, advancing various optimization and analysis goals.

Our framework design and experience demonstrates the feasibility of this technique, and also highlights the research challenges involved. Extracting and reconstructing data structure changes is itself a non-trivial effort, with further complexity provided by the need for appropriate, scalable representations. The two forms of visual output we describe attempt to accommodate different needs with respect to detail and large scale analysis, while still encoding useful information.

There are a great many potential future directions for this work. Our dynamic data, for instance, can be mapped to static code locations for direct comparison with static algorithms. This can help guide and measure static algorithm design. The efficacy of dynamic versions of other, more efficient if less precise data structure representations can also be evaluated.

Visualization improvements are many of course. We hope to improve animation quality by adapting existing tools to support custom, if sub-optimal incremental layout. Scaling concerns with such literal representations can be partially addressed through the use of interactive visualization techniques. Given the large data volume, however, novel visualizations that compactly summarize specific properties are more immediate goals.

## Acknowledgements

This research has been supported by the le Fonds Québécois de la Recherche sur la Nature et les Technologies and the Natural Sciences and Engineering Research Council of Canada.

## References

- [1] J. Bogda and A. Singh. Can a shape analysis work at run-time? In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.
- [2] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, New York, New York, United States, Nov. 2002. ACM Press.
- [3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java controller. In *PACT01*, pages 280–291, Barcelona, Spain, Sept. 2001.

- [4] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis and interprocedural techniques for automatic parallelization of C codes. *Int. J. Parallel Program.*, 30(1):37–63, 2002.
- [5] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master’s thesis, McGill University, Montréal, Québec, Canada, 2004.
- [6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’03)*, pages 149–168, 2003.
- [7] P. Fradet and D. L. Métayer. Shape types. In *POPL ’97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997.
- [8] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
- [9] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL ’96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996.
- [10] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL ’05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 310–323, New York, NY, USA, 2005.
- [11] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *IEEE Transaction on Parallel and Distributed Systems, Vol. 1, No. 1*, pages 35–47, January 1990.
- [12] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, 1992.
- [13] D. Johannes, R. Seidel, and R. Wilhelm. Algorithm animation using shape analysis: visualising abstract executions. In *SoftVis ’05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 17–26, New York, NY, USA, 2005.
- [14] N. Klarlund and M. I. Schwartzbach. Graph types. In *POPL ’93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 196–205, New York, NY, USA, 1993.
- [15] M. Leone and R. K. Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization. Technical Report No.490, Computer Science Department, Indiana University, Sept. 1997.
- [16] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E. Zapata. A new dependence test based on shape analysis for pointer-based codes. In *LCPC ’04: Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.
- [17] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV’01, First Workshop on Runtime Verification*, Paris, France, July-23 2001.

- [18] T. Printezis and R. Jones. GCspy: an adaptable heap visualisation framework. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343–358, New York, NY, USA, 2002.
- [19] S. P. Reiss and M. Renieris. Jove: Java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 115–124, New York, NY, USA, 2005.
- [20] N. Røjemo and C. Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 34–41, New York, NY, USA, 1996.
- [21] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 12–17, New York, NY, USA, 2000.
- [22] SPEC Corporation. The SPEC JVM Client98 benchmark suite. <http://www.spec.org/jvm98/jvm98/>, 1998.
- [23] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.
- [24] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.