Evaluating Lossy Collections for Java Applications

Jeremy Singer   David R. White
School of Computing Science
University of Glasgow
{jeremy.singer,david.r.white}@glasgow.ac.uk

Abstract

We propose to remove live objects from near-full heaps to reduce memory pressure. We modify Java Collections to enable lossy behavior. Some DaCapo benchmarks tolerate an amount of live data loss.

Keywords memory management, Java collections, lossy data structures

1. Introduction

At present, a typical memory-managed application with live data that exceeds the maximum runtime heap size will crash with an `OutOfMemory` error. The alternative we propose in this paper is for the application to demonstrate *graceful degradation* as it approaches a limiting factor, such as the heap size limit; we trade-off heap space for computational accuracy.

Suppose a program P_0 normally executes in a minimum heap size of H_0. We want to enable an *approximate* version of the program P_1, such that P_1 executes in a heap size H_1 where $H_1 < H_0$, although P_1 might result in degraded output relative to P_0. Furthermore, we wish to create P_1 at run-time without the intervention of the programmer.

Our key insight is that 'not all data is created equal'. If we can identify data that is non-critical and less valuable in some quantifiable sense, then we can opportunistically discard some of this data. Existing programming language concepts such as weak references implicitly acknowledge that some heap-allocated objects are less important and may be garbage collected if there is little heap space available. High-level libraries such as software caching frameworks follow a similar principle (Google 2014).

```
public static <E extends Comparable<? super E>>
              void heapsort(E[] array) {
  PriorityQueue<E> heap;
  heap = new PriorityQueue<E>(array.length);
  for (E e : array) {
    heap.add(e);
  }
  for (int i=0; i<array.length; i++) {
    array[i] = heap.remove();
  }
}
```

Figure 1. An Example Heapsort using Java Collections

Our main research question is: *how can we trade-off space and accuracy automatically, without programmer intervention?*

1.1 Contributions

The contributions of this paper are as follows:

1. We describe a *motivating scenario* for approximate computing driven by runtime memory management.
2. We demonstrate how approximation can be supported transparently in Java applications by *library-level modifications*.
3. We sketch *policies* to describe the characteristics of several concrete instances of approximate Java collections.
4. We apply these policies to standard benchmarks to *evaluate the potential* of this idea.

1.2 A Simple Example

To clarify our proposal, consider the program in Figure 1, a simple heapsort algorithm found on the web[1]. If available memory is exhausted before the priority queue is fully populated, then the code will give an `OutOfMemory` error and return no output. If instead we employ our approximation method described in Section 2, then we sacrifice accuracy in return for continuing execution which results in a partially correct output; it is worth noting that approximate collections are only useful where *a partial output is better than no output at all*. As our technique sometimes uses substitution

[1] `http://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Heapsort#Java`

to replace objects that are not retained, the level of error introduced may depend on the amount of redundancy in the input data.

Figure 2 illustrates the trade-off between the level of data loss and the accuracy of the resulting output of the heapsort. We consider each missing item that should have appeared in the resulting array as an error. Figure 2(a) shows that, with no redundancy in the data, the number of errors introduced grows linearly. Figure 2(b) shows that introducing a certain amount of redundancy in the data causes a non-linear relationship, and much lower error for most levels of data loss.

Although this is a trivial example, it demonstrates that much of the data stored by a program in standard data structures can be substituted without causing the program to terminate, and that induced errors are proportional to the amount of data sacrificed. Where the lossy data structure corresponds to a buffer for pending work, we may gracefully degrade the operation of the program by discarding some data.

2. Approximate Memory Management Policies

As we discard live data, we aim to minimize the impact of data loss on the quality of a program's output.

2.1 Policies

There are four major decisions to be made by any data discard policy:

1. When do we delete data?
2. How much data should we delete?
3. Which data should we delete?
4. How should we attempt to mitigate the effects of deletion?

We do not address the first two questions directly in this paper, other than to state that we assume some upper limit for memory exhaustion, and that an exponential function should specify the relationship between the current distance to that limit and the amount of data being deleted, i.e. as the limiting resource becomes increasingly exhausted, the probability of deleting data approaches 1.

As to *which* data we should delete, we focus on objects that are referenced by instances of the *Java collections framework*. We hypothesise that collections are used principally as container data structures for Java applications, and that they are more likely to contain data to be processed rather than control information; the latter is critical to execution, whereas the former is not. To ensure that we target only the 'hot' collections turning over the majority of data, we impose a threshold such that we only delete from collections that experience more than n insertions or additions, or those that contain more than a given threshold size of data.

We mitigate the effects of deletion through two methods: First, we select data *stochastically*, i.e. we do not always delete the same data. This prevents pathological situations where the use of an approximate data structure would always result in the same undesired behaviour. Second, we employ *substitution*, to replace deleted objects with existing objects of the same concrete type.

2.2 Deletion Schemes

Once we have decided to delete data from a collection data structure, there are several points during program flow where we could perform a deletion. We choose to remove data at insertion time.

Our deletion scheme is randomised to avoid the pathological cases that arise from anything more regular. For example, an alternative would be to delete items periodically. Stochastic deletion allows for a checkpointing and restarting approach to failure recovery, and it also makes for an elegant and efficient decision process; we simply increase the probability of deletion as the limit of available memory is approached.

We rely on the garbage collector to remove objects from the heap; we are only responsible for dropping references to them within a collection. It is an implicit assumption of our approach that items added to a collection will not be referenced elsewhere, or only within another collection.

2.3 Recovery Schemes

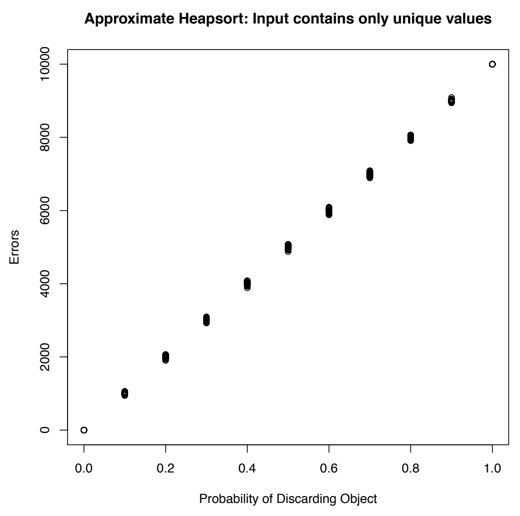
Rather than attempt to repair the user program, for example through injecting null pointer checks, we simply modify the behaviour of the collection itself. The first option to reduce the impact of a deletion is to do nothing at all; a strategy that works surprisingly well for some programs, see Section 3.1. Alternatively, we could either *substitute* or *synthesise* deleted data. Synthesis requires intercepting retrieval API calls and returning new objects. This is, however, problematic in the sense that the collection must then keep track of what data has been deleted; is this `null` return value from a `HashMap.get(...)` call a true negative or a false negative? How do we synthesise the data? Perhaps we would need to keep track of the types and other information about deleted objects. We do not pursue such an approach for now.

Instead, we usually choose to substitute data objects in place of those discarded. Provided other objects are present in the collection that are type-compatible with the deleted object, we simply insert another reference to the substitute object within the collection.

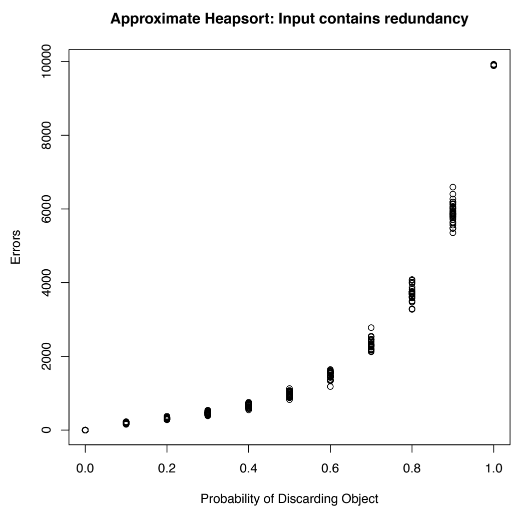
2.4 Implementation using AspectJ

Rather than invasive modification of Collections library code or the underlying VM, we use aspect-oriented programming techniques (Kiczales et al. 1997) to modify runtime behavior.

Effectively, the stochastic behaviour we have induced in the Collections data structures is a *cross-cutting concern*, to



(a) No redundant data, initial array is permutation of the sequence 0...9999



(b) Redundant data - initial array contains integers uniformly drawn from (0,100)

Figure 2. Probability of discarding objects versus induced errors in heapsort, for arrays of size 10,000 with and without redundancy. 30 points for each level of probability value are shown.
.

which client code should be oblivious. We specify pointcuts for methods in high-level abstract classes like `Map` and `List`. At weave-time, these pointcuts will match instances of concrete subclasses like `HashMap` and `ArrayList`.

The pseudocode in Figure 3 gives an example of the advice we apply. Further details are available in our online repository (White and Singer 2015). Note that AspectJ is used for convenience in prototyping our approximate data structure concepts. Aspect-orientation incurs runtime overhead that could be mitigated by closer library or VM integration in a production system.

```
procedure ADD TO A COLLECTION($collection, input$)
    Increment $insertionCounter$ for $collection$
    if $collection$ is not empty then
        if $insertionCounter > threshold$ then
            Set $discard$ true with probability $p_{drop}$
        end if
    end if
    if $discard$ then
        for $o$ in random permutation of $collection$ do
            if $type(o) == type(input)$ then
                Replace $input$ with $o$
                Exit loop
            end if
        end for
    end if
    return proceed($collection$,$input$)
end procedure
```

Figure 3. Aspect Pseudocode for a Simple Substitution Policy, i.e. Collection.add / other / p_{drop} / $threshold$

3. Experiments

In this section, we study the application of our approximate Collections framework to the DaCapo Java benchmark suite (Blackburn et al. 2006). We intercept calls to Java Collections methods in order to enforce various approximate memory policies, as outlined in Section 2.2.

Table 1 records an initial limits study, to measure the prevalence of Java collections usage in DaCapo. We use a custom JVMTI agent[2] which measures the proportion of heap-allocated memory that is reachable through live Java collections instances immediately after GC. The table shows the mean proportion of the heap reachable through live collection instances, over the first two minutes of execution of the DaCapo benchmarks with `small` inputs.

3.1 Success Rate Curves

We will describe the approximation policies we apply to the benchmarks in terms of (i) the collection update method we intercept, (ii) the replacement value we use for the update, (iii) the probability of interception, and (iv) the data structure size threshold above which we might apply interception. For instance, the policy `Map.put` / `null` / 0.1 / 100 means that we intercept `put` methods to all objects that are concrete subclasses of `Map`, we associate the key with `null` rather

[2] available in our repository (White and Singer 2015)

benchmark	*reachable data /KiB*	*live size /KiB*
avrora	2021	2431
batik	3626	3931
eclipse	8114	8593
fop	9227	10244
luindex	1168	1244
lusearch	1446	2451
pmd	1178	1264
sunflow	3035	3873
tomcat	4701	5116
xalan	1609	4918

Table 1. Mean heap-allocated memory reachable through collections objects, measured immediately after GCs

than the supplied value, the probability of this 'forgetful' action is 0.1, for Map objects that have at least 100 bytes of information stored in them already. Another example: `List.add` / other / 0.01 / 1000 means that we intercept `add` methods to all objects that are concrete subclasses of `List`, we store a reference to another value that is already in the list, so long as it has the same concrete type as the value that should have been written, the probability of this 'forgetful' action is 0.01, for List objects that have at least 1000 bytes already stored in them. When we apply these policies to the DaCapo benchmarks, we investigate a range of probabilities and storage size thresholds.

The DaCapo benchmark harness includes a validation check for each benchmark execution. This stage compares checksums for the standard output and error streams against reference checksums for correct benchmark behaviour. We use this validation to measure successful execution. For each policy, we run 30 separate executions of a single iteration of the benchmark with the `small` input size. We record how many of the 30 executions pass validation. This measure of success does not necessarily mean that the benchmark application completes successfully, merely that no errors were visible on its recorded output streams.

Figures 4 and 5 show some benchmark responses to the policies applied. We have focused on `List` and `Map` objects at present, given their prevalence in the DaCapo benchmarks.

We observe different kinds of benchmark response:

1. **Polarized** response: the benchmark either always succeeds or always dies, for a particular policy (e.g. lusearch, jython).
2. **Gradated** response: the benchmark sometimes succeeds when increasing amounts of data is forgotten. There is a *tradeoff curve* for data loss and benchmark success (e.g. avrora, luindex).
3. **Failed** response: the benchmark never succeeds. Some benchmarks do not interact well with our AspectJ instrumentation (e.g. tomcat, xalan).

Benchmarks that exhibit the gradated response are appropriate for approximate collections. Some tuning may be required to find the limits of the tradeoff region. The benchmark response graphs show that applications exhibit different behaviour for different data structures and for different replacement policies.

In a small number of cases, e.g. batik in Figure 4, the success rate increases as more data is discarded. This is a characteristic of *worklist* style algorithms, when throwing work away (especially if it is potentially corrupted work due to data loss) makes the overall benchmark more likely to succeed without errors.

3.2 Fuzzy Output Tolerance

Some applications have an inherent tolerance to noisy data processing. These applications will be able to continue execution and produce sensible output even when internal data structures are modified and data may be lost. Whereas in Section 3.1 we only considered successful execution to be output streams corresponding to the reference outputs—here we are interested in output that is *partially* correct, yet still of some value.

Our example application is the batik program from the DaCapo benchmark suite. Batik is a scalable vector graphics (SVG) processing tool. The particular behaviour is rendering a bitmap image from a textual description of the vector graphics in XML format. As the image is rendered, each component part of the vector image is added to a worklist (`AbstractSVGList. item(SVGItem item)` for rendering. When we apply an approximate aspect to remove items from the worklist, this has the effect of losing parts of the rendered bitmap image. Figure 6 shows how the output bitmap file is affected by various levels of drop-rate. As the likelihood of dropping data increases, the image loses sub-shapes. Some of the runs fail with `IndexOutOfBoundsExceptions`—it may take several runs before a single output file is produced.

The key point is that this output file (when we can generate it) is of some value to end-users, even if data is missing. Humans may infer missing information (i.e. black regions in the map). At other times, the missing information (statistics about local populations) may not be required by the user. So there is partial value in incomplete output. This inbuilt error tolerance for human consumption is generally characteristic of visual and audio outputs.

4. Related Work

Nguyen and Rinard (Nguyen and Rinard 2007) demonstrate the use of *circular buffers*. They perform a profiling-based analysis to determine the maximum number of distinct objects alive from each allocation site. Then they transform the program to use a circular buffer of size N at allocation site s when they show there never more than N live objects at s. They proceed to reduce the sizes of the circular

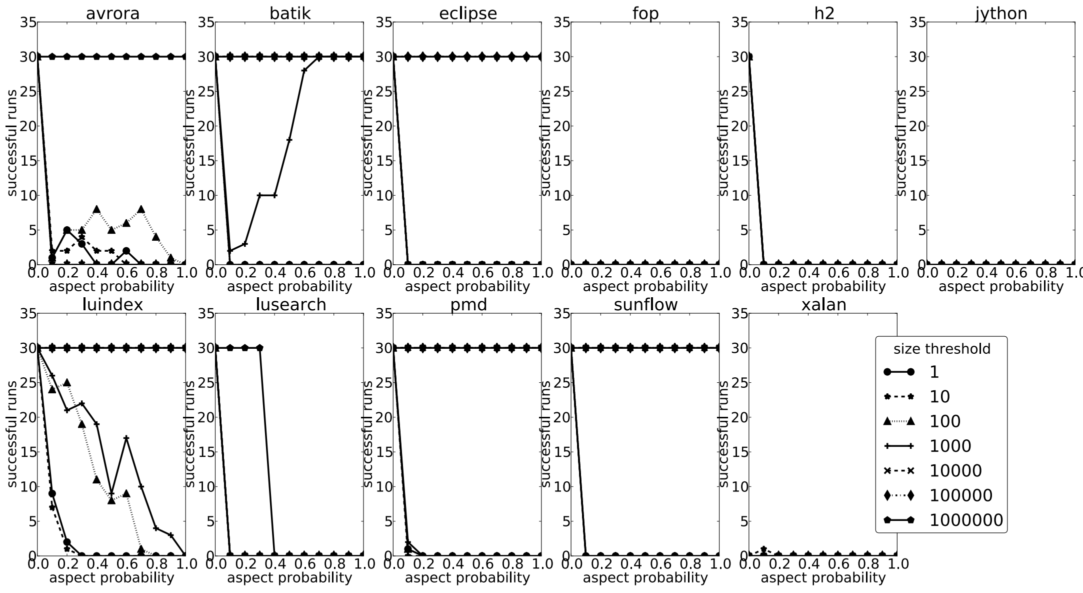


Figure 4. Successful executions (out of 30 runs) for DaCapo benchmarks with the List.add/`null`/p/s policy for various probabilities p and list sizes s

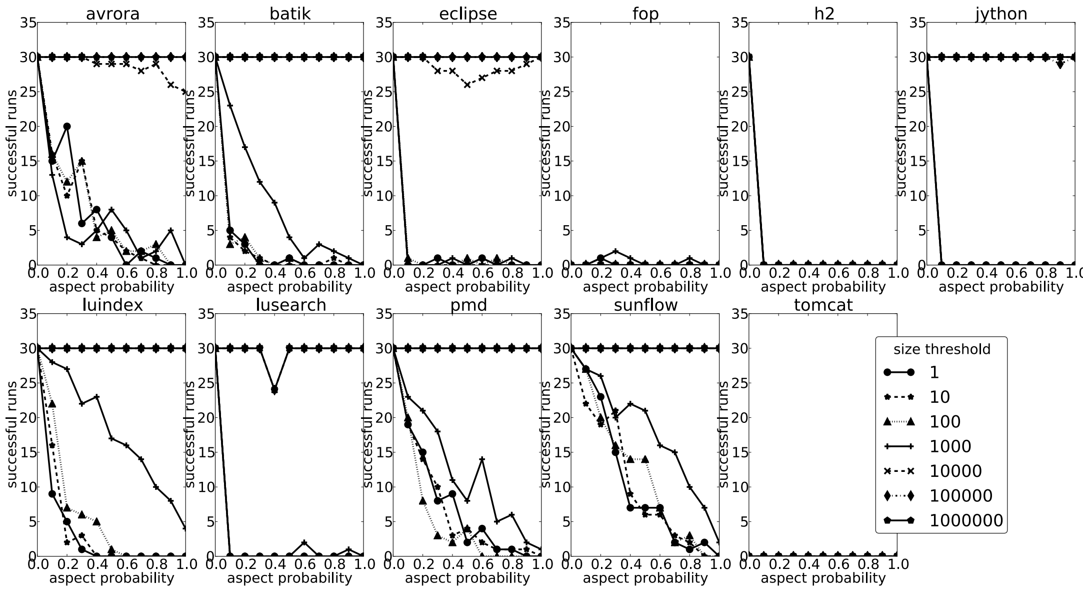


Figure 5. Successful executions (out of 30 runs) for DaCapo benchmarks with the Map.put/other/p/s policy for various probabilities p and map sizes s

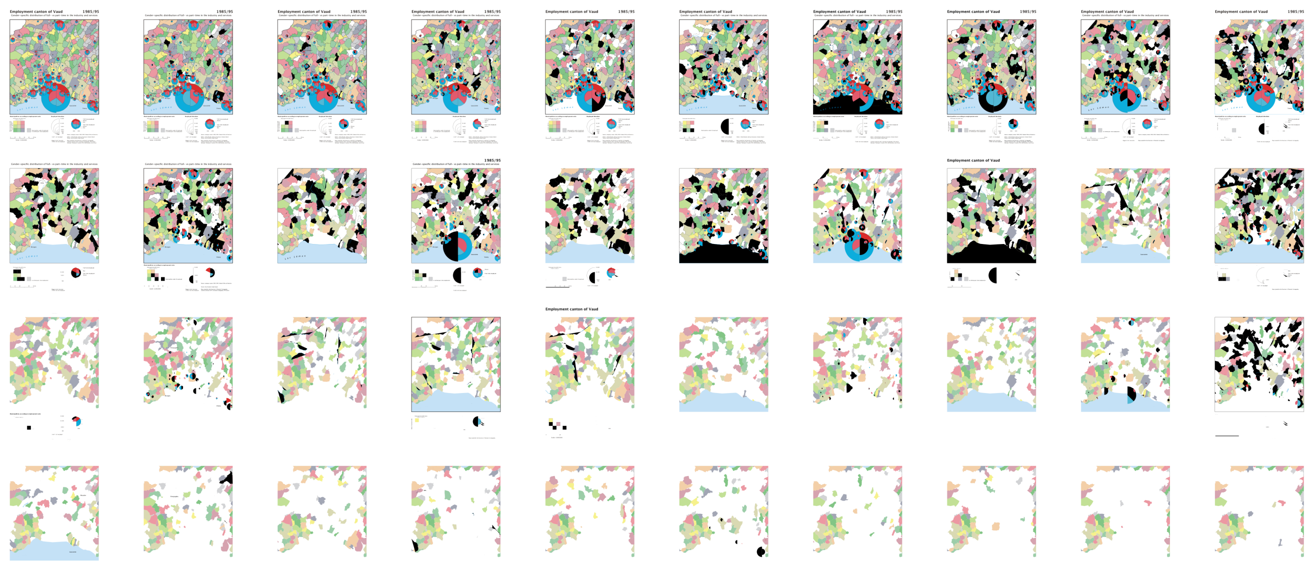


Figure 6. A series of bitmap output files generated by the batik DaCapo benchmark with `small` input.. From left to right, top to bottom, the probability of dropping data increases by 0.025 between adjacent images, from 0.0 to 0.975 inclusive. The approximate memory policy intercepts `list.add()` with 10kB size threshold and inserts `null` values.

buffers, thus overwriting live objects. (This is equivalent to our *replacement* policy in Section 2.4 above.) They report that, for several widely used programs such as the FreeCiv game server and the Pine email client, there are no observable behavior anomalies due to object replacement in circular buffers during prolonged periods of typical user interaction. Nguyen and Rinard apply this technique to run programs that would otherwise suffer from memory leaks. They refer to this general approach as *failure-oblivious computing*. Rinard discusses this notion further (Rinard 2003). Rinard et al. (Rinard et al. 2010) outline a generalized framework to show how optimizations of this kind fit abstract patterns for approximate computing—situations where accuracy can be traded off for some other reduced resource.

Matias et al. (Matias et al. 1994) present an approximate version of a specific data structure, which denotes ordered multisets of integers. They use this data structure for various dynamic programming applications including minimum spanning tree and single-source shortest path. They abstract integer values to their most-significant bits only, which reduces problem to a smaller domain. They give a theoretical treatment, analysing reductions in computational complexity, and discussing implementation concerns on a hypothetical random access machine model. Chazelle (Chazelle 1998) introduces an approximate version of priority queues, which he calls *soft heaps*. Keys are occasionally corrupted (their values are increased) to facilitate moving items in groups when updating the heap. Chazelle shows reductions in asymptotic complexity for certain operations with this approximation. In contrast, we present a generalized approximation framework that is oblivious of the structure and semantics of particular collection types.

5. Conclusions & Future Work

Our current work is entirely preliminary; the next stage is to develop our ideas further through wider application of this method to more libraries and a wider range of benchmarks. From initial experience, we think it is inevitable that programmer intervention will be required for some applications to degrade gracefully. In particular, we envisage situations where data critical to successful program completion could be inadvertently lost. To remedy this, we propose the development of debugging tools that will automatically identify the cause of undesirable behaviour, paired with source code annotations to identify essential data that should not be considered for deletion.

We also intend to investigate why some applications respond better than others to the approximations. The response may be dependent on the application, and on the deletion/replacement policy.

Acknowledgments

We thank Martin Rinard and Richard Jones for their helpful feedback on this work.

References

S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*. ACM, 2006.

B. Chazelle. Car-pooling as a data structuring device: The soft heap. In *Proc. European Symposium on Algorithms*, pages 35–42, 1998.

Google. Google guava cache library, 2014. URL `http://code.google.com/p/guava-libraries/wiki/CachesExplained`.

G. Kiczales et al. Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242, 1997.

Y. Matias, J. S. Vitter, and N. E. Young. Approximate data structures with applications. In *Proc. SODA*, pages 187–194, 1994.

H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proc. ISMM*, pages 15–30, 2007. .

M. Rinard. Acceptability-oriented computing. *SIGPLAN Notices*, 38(12):57–75, Dec. 2003. .

M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Proc. OOPSLA*, pages 806–821, 2010.

D. R. White and J. Singer. Online project repository, 2015. github URL to be supplied.