Heap Profiling for Space-Efficient Java

Ran Shaham
Tel-Aviv University and
IBM Haifa Research
Laboratory
rans@math.tau.ac.il

Elliot K. Kolodner
IBM Haifa Research
Laboratory
kolodner@il.ibm.com

Mooly Sagiv
Tel-Aviv University
sagiv@math.tau.ac.il

ABSTRACT

We present a heap-profiling tool for exploring the potential for space savings in Java programs. The output of the tool is used to direct rewriting of application source code in a way that allows more timely garbage collection (GC) of objects, thus saving space. The rewriting can also avoid allocating some objects that are never used.

The tool measures the difference between the actual collection time and the potential earliest collection time of objects for a Java application. This time difference indicates potential savings. Then the tool sorts the allocation sites in the application source code according to the accumulated potential space saving for the objects allocated at the sites. A programmer can investigate the source code surrounding the sites with the highest savings to find opportunities for code rewriting that could save space. Our experience shows that in many cases simple code rewriting leads to actual space savings and in some cases also to improvements in program runtime.

Experimental results using the tool and manually rewriting code show average space savings of 18% for the SPECjvm98 benchmark suite. Results for other benchmarks are also promising. We have also classified the program transformations that we have used and argue that in many cases improvements can be achieved by an optimizing compiler.

Keywords

Compilers, garbage collection, Java, memory management, profiling, program analysis

1. INTRODUCTION

GC does not (and in general cannot) collect all the garbage that a program produces. Typically, GC collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again, even though they are reachable. This is illustrated pictorially in Figure 1.

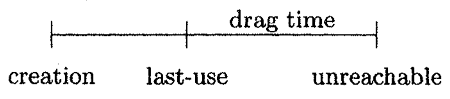

Figure 1: The lifetime of an object.

As shown in Figure 1, and following Röjemo and Runciman [21], we refer to the time interval from the last use of an object until it becomes unreachable (and thus can be collected by a reachability-based GC [28]) as the object's *drag time* and to object itself as a *dragged object*. Drag time measures a potential for space savings independent of the actual GC method or GC implementation.

For a previous paper [25] we measured the drag time for objects and tried to characterize the dragged objects. The results showed a potential for space savings from 23% to 74% for the SPECjvm98 benchmark suite if objects could be collected at their last use. In this paper, we describe a prototype tool based on drag measurements that is aimed to help in actual reduction of space. Applying our tool, we are in fact able to save on average 18% of the space for the SPECjvm98 benchmarks.

Our measurements show an average of the potential space savings when GC is performed on the entire heap. The actual savings will depend on factors such as the size of the heap (which influences the frequency of GC) and the type of GC (e.g., generational GC). Since our techniques reduce the set of reachable objects, space savings are expected for all JVMs employing reachability-based GC.

1.1 Drag Reduction Tool

The tool acts in two phases: (i) measure the dragged objects, and (ii) analyze the results and produce a list of allocation sites of dragged objects sorted by their potential space savings.

For the first on-line phase, we instrumented Sun's JVM 1.2 [13] in order to measure dragged objects. Specifically, we record at regular intervals (i) the objects reachable at the end of the interval and (ii) the objects that are also used subsequent to the interval. We call the former the *reachable* objects and the latter the *in-use* objects.

Our goal is reducing the size of the reachable heap at every sampling point, by reclaiming reachable objects that are not in-use. Thus, as a second off-line phase, we partition the dragged objects according to their allocation site, and for each allocation site we sum the *drag space-time product* of every dragged object. The drag space-time product, or *drag* for short, is the product of the size of the object and the time the object is reachable but not in-use. Allocation sites having a large drag suggest a potential for significant space savings. Therefore, our tool sorts allocation sites according to their drag.

1.2 Reducing the Drag

Our off-line profiler tool can be used either directly by a programmer or to produce input for a profile-based optimizer. Currently, we use the tool to direct manual code inspection to the allocation sites having a large drag. Examining these sites we have found that three kinds of simple correctness preserving program transformations have been effective in achieving space savings: (i) explicitly assigning null to a dead reference to an object, (ii) removal of dead code, and (iii) lazy allocation of objects. These program transformations cannot harm the space consumption of a program, and in most cases save space.

In Section 5 we describe the static information that is needed in order to infer that the transformations can be automatically conducted. Not all of the transformations need to be performed on the source code; instead, some could be implemented in the runtime of the JVM, e.g., using stack-maps to maintain liveness information [1]. In order to reduce the cost of analyses, which in many cases involve whole program analysis, our profiling tool can be used to guide the analyzer to the parts of the code that are "drag-hot".

The tool was applied to nine benchmarks. Although we were not familiar with the code of these benchmarks, it took us just a few hours to understand the results produced by the tool and to rewrite the code. Also, only few lines of code had to be rewritten. Of course, we believe that an application programmer using our tool on his own code could do a better job, both in terms of speed (of rewriting) and more importantly in terms of space (reduction of drag).

Code rewriting for the benchmarks we considered reduces the total drag by 51% on average, leading to an average space saving of 15%. In some cases the runtime cost is also reduced. This is mainly due to: (i) GC is invoked less frequently and (ii) allocation and initialization are avoided for objects that are never used.

1.3 Related Work

Röjemo and Runciman performed similar measurements for Haskell, a lazy functional language [21]. They demonstrate how to save space by rewriting the code of one benchmark, a Haskell compiler. However, their rewritings are based on a deep understanding of the code. Other tools for memory profiling [23, 19] show the heap configuration and allocation frequency; they also help in tracking memory leaks by allowing the heap to be inspected at points during the execution of the program. As noted by Serrano and Boehm [23], tracking memory leaks by inspecting dragged objects is orthogonal to tracking leaks by inspecting the heap during the course of execution.

Automatic techniques aimed at reclaiming more storage include integrating liveness analysis and GC [1] and compile-time GC. Agesen *et al.* [1] integrate liveness analysis of local reference variables with GC, so that reference variables with no future use (i.e., dead references) are not regarded as part of the root set. In an earlier paper [24], we show how to integrate liveness analysis for arrays of references with GC, so dead array elements are not regarded as part of the root set. Turning to compile-time GC, most work has been done for functional languages [2, 15, 8, 16]. However, escape analysis [3, 27], which is in some sense a special case of compile-time GC, has been recently applied to Java. Experimental results in [3] show an average speedup of 21% using escape analysis to stack allocate objects.

1.4 Outline of the Rest of this Paper

The remainder of the paper is as follows. Section 2 describes our prototype tool. Section 3 presents the framework of the experiment. In Section 4 results are discussed. Section 5 describes the static information necessary to automatically apply space savings transformations. We conclude in Section 6.

2. THE TOOL

The heap-profiling tool consists of two phases. In the first phase, an instrumented JVM runs a Java application and outputs information on dragged objects to a file. This is the same instrumentation reported in our previous work [25]. The second phase analyzes the file, producing a list of allocation sites sorted by their potential for drag reduction. We found that it is best to have information regarding the *nested allocation site* of a dragged object, i.e., the call chain leading to the allocation.

2.1 The Instrumented JVM

At a high level we associate a time field with every object in order to record drag information. On every use of an object, we record the current time in its field. Thus, the field always holds the time at which the object was last used. In order to approximate the time at which an object becomes unreachable, we frequently trigger GC; thus when an object is collected, its drag time is calculated as the difference between the approximated time it becomes unreachable (i.e., its collection time) and the time it was last used. We measure time in bytes allocated since the beginning of program execution, assuming that all uses of an object in the interval between consecutive garbage collection cycles are performed at the beginning of the interval.

The instrumented JVM is based on Sun's JVM 1.2 (aka classic JVM). Its memory system uses indirect pointers to objects ("handles"), thus objects can be relocated easily during GC. Below we provide greater detail on our instrumentation.

2.1.1 Implementation

We attach a trailer to every object to keep track of our profiling information. We do not count the space taken for this trailer in our data. The information in an object's trailer is written to a log file upon reclamation of the object or upon program termination. An object's trailer fields include its creation time, its last use time, its length in bytes, its nested allocation site and its nested last-use site. The length includes the header and the alignment (i.e., the bytes that were skipped in order to allocate the object on an 8 byte boundary), but excludes the handle and the trailer.

Object information is updated upon the following events:

Object Creation The creation time, length and nested allocation site are set. The level of nesting can be set in order to tradeoff more accurate information and speed.

Object Use The last use time and nested last use site are set. The following events constitute an object *use*: (1) getting field information (e.g., via `getfield` bytecode), (2) setting field information (e.g., via `putfield` bytecode), (3) invoking a method on that object (e.g., via `invokevirtual` bytecode) (4) entering or exiting a monitor on that object (via `monitorenter`, `monitorexit` bytecodes) and (5) derefencing a handle to that object. The last item is relevant for native code, since manipulating a Java object in native code is done through a handle to the Java object.

After every 100 KB of allocation we trigger a *deep GC* (a larger interval yields less precise results). A deep GC consists of the following steps: (1) GC, (2) run finalizers for all objects waiting for finalization, (3) GC. Forcing finalization ensures instant reclamation of all unreachable objects and removes a source of non-determinism (since finalization would otherwise occur in a separate thread). When an object is freed, we log all of the information collected in its trailer. When the program terminates, we perform a last deep GC and then we log information for all objects that still remain in the heap.

The rules for the collection of `Class` objects are not the same as for regular objects. Thus, we exclude them and the special objects reachable from them (e.g., *constant pool* strings and per-class security-model objects) from our reports.

2.2 Drag Reporting

The second phase analyzes the information written to the log file by the first phase.

The drag of an object is the product of the size of the object and the time the object is reachable and not in-use. We partition the dragged objects into groups according to their nested allocation site, associating with each group the sum of the drag for all objects allocated at its site. Sometimes an allocation site is used in many contexts and a large drag may be distributed among several smaller drag groups when partitioning solely according to nested allocation site. Thus, we also do a coarse-grained partition according to the allocation site.

Sometimes the site at which a dragged object was last used (hereafter, called last-use site) may hint at the code rewriting strategy best-suited for reducing its drag. For example, if a dragged object remains reachable due to dead references, the last-use site may hint at the program point where a reference to that object becomes dead. Thus, we also partition dragged objects according to nested allocation site and last-use site. In `Euler` benchmark (see Table 1), the last-use site is used to determine which reference variable prevents an object from being reclaimed.

A special case for a dragged object is an object that is never used. We partition these *never-used* objects according to nested allocation site and according to allocation site. Our experience so far suggests that a large drag caused by never-used objects is a "sure bet" for code rewriting.

Graphs showing the amount of heap memory in-use and the amount reachable over time can also be produced as reported in our previous work [25]. These are useful for visualizing the overall memory usage of an application.

3. APPLYING THE TOOL

We applied the drag reduction tool to sample Java programs to determine the allocation sites in those programs contributing the most to the drag. Then we carefully analyzed the source code around those sites in order to find program transformations that reduce the drag. Interestingly, we found out that a very shallow understanding of the program suffices to reduce a significant part of the drag. Moreover, future-optimizing compilers may be able to automatically conduct these transformations (see Section 5).

In this section we describe the drag reducing program transformations and how we used the tool to direct the choice of the proper transformations.

3.1 The Benchmark Programs

Specifically, we applied the tool to nine sample Java programs. We list the programs in Table 1. The second and third columns show the total number of application classes and the total number of application source code statements, respectively. JDK classes and general SPEC classes shared by all SPECjvm98 benchmarks are not included in Table 1. There are 32 general SPEC classes having total of 3173 source code statements.

We employed five of the benchmarks from the SPECjvm98 benchmark suite [26]. We do not consider `compress` or `mpegaudio` because they do not use significant amounts of heap memory. Another two benchmarks were taken from Java Grande benchmark suite [12]. The last two benchmarks are internal IBM tools.

3.2 Reducing the Drag

The code was manually investigated in order to determine the reasons for drag. Since we were unfamiliar with the code of the sample programs, we employed JAN [20], a tool for analyzing and understanding Java programs. We used two kinds of information provided by JAN: (i) the *class hierarchy graph*, and (ii) the *program call graph*. This kind of information is also provided by many other Java compilers and analysis tools.

We used the class hierarchy graph for accelerating source browsing, e.g., locating overloaded methods. We used the

Benchmark	Classes	Stmts	Short Description
javac	176	12345	java compiler
db	3	512	database simulation
jack	56	5106	parser generator
raytrace	25	1479	raytracer of a picture
jess	151	4567	expert system shell
mc	15	880	financial simulation
euler	5	726	Euler equations solver
juru	38	2505	web indexing
analyzer	258	35489	mutability analyzer

Table 1: The benchmark programs.

program call graph to check the applicability and validity of program transformations. For example, assigning null to a dead reference variable requires inspection of every possible use of that variable. Using the program call graph, we can check that some uses of the variable, which appear in the source code, do not actually occur at runtime, e.g., if the graph shows that the method is never invoked.

Source transformations were applied only after a thorough inspection of the source code and validation that the transformation was applicable using the program call graph. We also checked that the original and revised benchmarks produce identical results on several inputs. In order to measure the effect of code rewriting on the running times of the programs, the original and revised versions were invoked and their running times also compared.

The code of the benchmarks as well as selected classes of the JDK itself were rewritten using the code rewriting strategies discussed below in Section 3.3. The tool was reapplied to the revised code in order to measure the resulting drag, and determine the actual space savings. Sometimes, the results revealed more opportunities for drag reduction; in that case, another cycle of code rewriting and applying the tool took place.

3.3 Code Rewriting Strategies

Surprisingly enough (at least for us), it appears that a few simple code rewriting techniques suffice for reducing much of the drag. The first technique is simply assigning the null value to a reference that is no longer in use. This reference may be a local variable, an instance field, a static field, or an element within an array of references. The second technique is removing dead code, i.e., code that has no effect on the result of the program. Our main interest is in dead code that produces dragged objects. The third technique is delaying the allocation of an object until its first use; thus, avoiding memory consumption for objects that are never used. In Section 5 we discuss how these code rewriting techniques can be replaced by automatic means.

3.3.1 Assigning Null

In many cases a dragged object remains reachable after its last use due to a dead reference. We inspect the code for all possible uses, identify the places where the reference becomes no longer used, and insert a statement assigning null to that reference. The details depend on the kind of variable containing the reference.

- For a local reference variable we inspect the containing method for possible uses.
- For an instance field we inspect the code according to visibility modifiers of the field, e.g., for a private field we inspect only the code of the containing class. Sometimes the program call graph is used to determine the context for a possible use. As noted earlier, we use program call graph information to invalidate possible uses in unreachable methods.
- For arrays of objects we have found cases where an element becomes dead. For almost all cases the array is being used to implement a data type similar to a Java vector and an element is being removed from the array [24].

Our tool provides information about the last line of code at which an object is used. This helps to find the place in the code where the object is no longer in use.

3.3.2 Dead Code Removal

Dead code [18] does not affect the result of the program. Using a feature of the tool showing objects that are allocated but never used, we find allocation sites where all objects are never-used. These never-used objects may be referenced by local variables, instance variables and array elements of type reference. We eliminate the allocation of these objects. This optimization is not always possible as it also removes the invocation of the constructors for these objects. We must guarantee that the constructor is the only code that references the object and that the constructor has no influence on the rest of the program, e.g., it does not update other objects or static variables and it cannot throw an exception for which there may be a handler in the surrounding code.

3.3.3 Lazy Allocation

Using the same feature of the tool described above, we have found allocation sites that produce many never-used objects. If these objects are big contributors to the drag, then we change the code to allocate them lazily. In particular, we eliminate the original allocation of the object and the variable that would have referenced the object remains null or is assigned null. Then, at every possible first use of the object, there is a test to check whether the variable is still null. If so, the object is allocated. We find possible first uses in the source code by employing the program call graph.

In general determining that postponing the allocation of an object does not change program semantics is hard. Thus, we only employ this transformation for the easy cases. In particular, the constructor may not depend on program state, e.g., it must have no parameters or parameters that are constant and it may not read program state (for example, access a static variable) in its body. Also, the constructor may not throw exceptions for which there may be handlers in the surrounding code. For all of the objects for which we applied this transformation, the only possible exception was `OUT_OF_MEMORY` for the allocation itself; thus, we only had to check that there were no handlers for `OUT_OF_MEMORY` in the program.

Lazy allocation may add runtime overhead for the checks at every possible first use. Thus, there is a risk that it could be detrimental for some program inputs, and care needs to be taken to apply it only when most of the objects allocated at the dragged allocation site are never-used. Nevertheless, this transformation has a lot of potential for reducing space and drag. We used it for just one of the benchmarks, `jack`.

3.4 Putting It All Together

Given the output of the drag-profiling tool we need to determine which program transformation to apply. Our experience shows that the first step is to find the method and the reference variable on which attention should be focused. We choose a nested allocation site with high drag. The bottom level is likely to be an allocation site in JDK or other library code, e.g., allocating a character array in `java.util.String`. We follow the call chain upwards looking for the first place in application code where a reference to the allocated object (or to an object containing the allocated object) is stored in a variable. We call this place the *anchor allocation site.* We call the variable the *anchor variable.*

Our experience suggests that the second step is to employ the output of the tool to investigate the lifetime characteristics of dragged objects at the anchor allocation site. In particular, the tool outputs the drag size due to objects allocated at a given anchor allocation site without any recorded use (the last use time is zero). The tool also partitions the dragged objects at that anchor allocation site according to their drag time, in-use time, and collection time. We have identified the following patterns of behavior:

1. All of the drag at the site is due to objects that are never-used. In some cases the only use of an object may be in its constructor and its in-use time is very short; we also consider these as objects that were never used.
2. Most of dragged objects at the site are never-used.
3. Most of dragged objects at the site have a large drag.
4. The variance of the drag for the objects at the site is high.

Based on the lifetime pattern at the anchor allocation site, one of the drag reducing program transformations might be applicable. The first pattern suggests the dead code removal transformation. The second pattern suggests the lazy allocation transformation. The third pattern suggests that assigning null (to a dead reference) is the most applicable transformation.

The fourth pattern suggests that there may be no program transformation that might help. For example there may be a large repository of objects as in the `db` benchmark. A query on the repository leads to a use of an object. However, each query accesses only a small number of objects and the queries are spread out over the whole application. Nevertheless the repository and all objects in it need to be kept as the exact queries cannot be predicted in advance.

These patterns do not cover all of the possibilities. However, most repeated more than once and we found them to be useful for finding applicable program transformations. Below we provide examples from the benchmark programs for each of the transformations and try to relate them to the patterns. Sometimes, when applying these transformations, the value of the anchor variable is assigned to other variables. These variables also have to be considered when applying the transformation.

3.4.1 Assigning Null

In `juru` the largest drag for an allocation site is 25.94MB^2. Character arrays of 100K elements are allocated at this site and assigned to a local variable. Each of these arrays is in-use for 200KB of allocation and then in-drag for another 200KB until it becomes unreachable. Assigning null to this local variable after its last use eliminates this drag and leads to a 33% reduction in total drag for `juru`. The drag time of these objects is short, but the objects are large so these objects contribute significantly to the drag space-time product. This fits the third pattern.

3.4.2 Dead Code Removal

In `raytrace` benchmark there are 17 allocation sites with the same behavior: an object is allocated and assigned to an array element; the object's last use occurs during its initialization, which is done in its constructor. Thus, all objects allocated at these sites are considered never-used. Each of these allocation sites contributes 4.77MB^2 to the drag. This behavior fits the first pattern and we apply the dead code removal transformation.

With the help of the program call graph, we verify that these objects referenced by the array elements are never accessed outside their constructors. We also verify that the constructors do not have any side effects. Thus, the code for the allocation of these objects can be removed. This leads to a 45% reduction in total drag.

3.4.3 Lazy Allocation

In the `jack` benchmark, the three allocation sites producing the largest drag are all in the same constructor. More than 97% of the drag for these three allocation sites is due to objects that are never-used. This behavior fits the second pattern and we apply the lazy allocation transformation.

We turn to the constructor's code. One `Vector` and two `HashTable` objects are allocated at the allocation sites. References to each of these data structures are assigned to instance fields. These instance fields have package visibility, i.e., they are visible to all package members. Thus, we scan the package for possible uses. It turns out that all uses of these instance fields occur in their containing class. We eliminate the allocations and before every possible first use of one of the instance fields, we add a test to check whether the allocation has already been done. If not, the allocation is performed.

4. RESULTS

We present the results of applying our heap profiling tool to the benchmark applications: the savings in space and the savings in time.

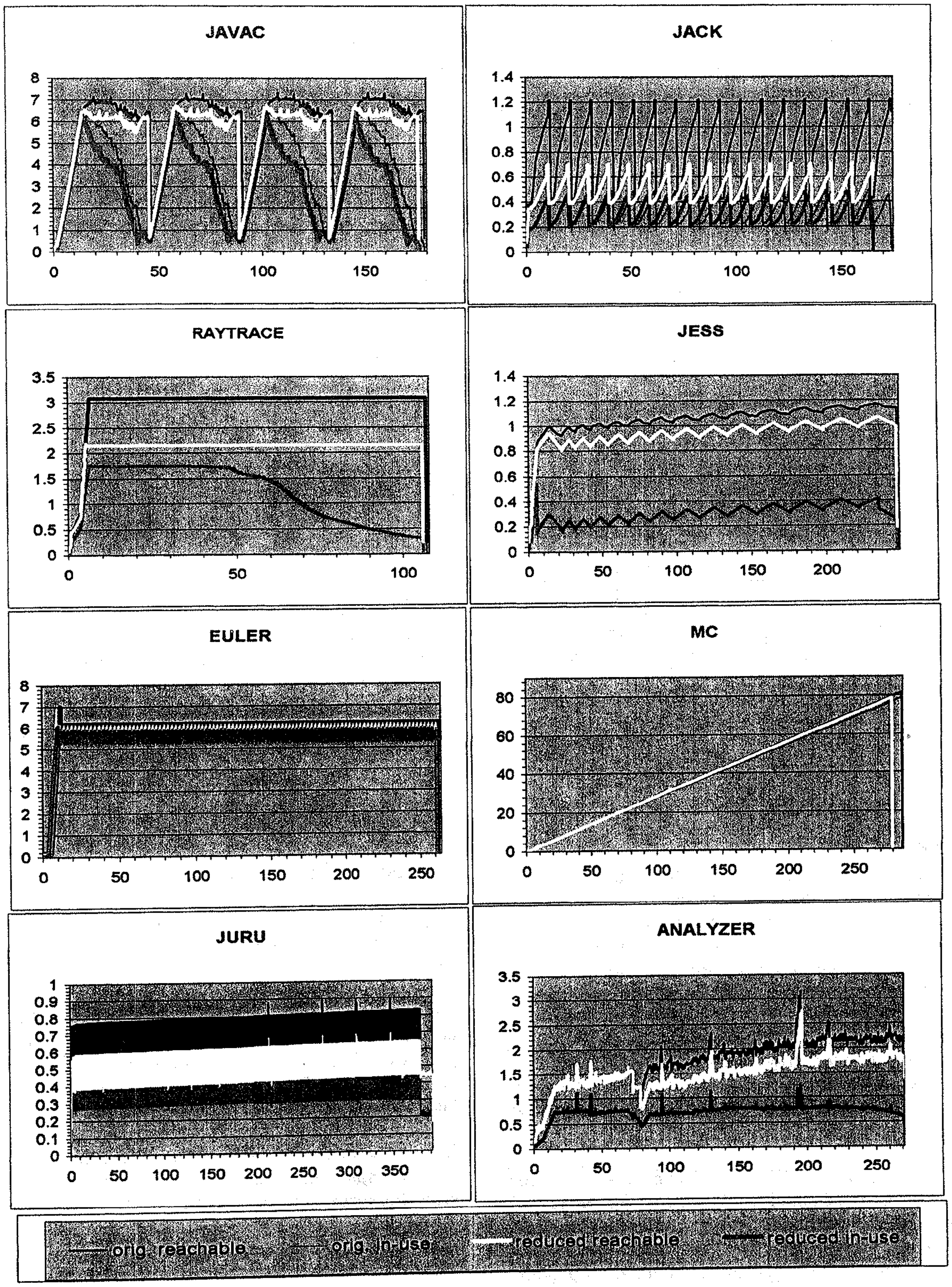

Figure 2: Original reachable/in-use heap size vs. revised reachable/in-use heap size. X-axis denotes allocation time in MB. Y-axis denotes size in MB.

Benchmark Program	Reduced		Original		Drag Saving Ratio (%)	Space Saving Ratio (%)
	In-Use Integral (M $Byte^2$)	Reachable Integral (M $Byte^2$)	In-Use Integral (M $Byte^2$)	Reachable Integral (M $Byte^2$)		
javac	566.49	937.09	656.19	1015.4	21.8	7.71
jack	50.58	82.24	57.07	141.93	70.34	42.06
raytrace	127.47	220.59	128.42	317.62	51.28	30.55
jess	74.01	231.91	73.67	260.86	15.47	11.1
euler	1421	1459.64	1424.34	1574.28	76.46	7.28
mc	10969.61	11010.44	11310.73	11747.09	168.82	6.27
juru	159.83	210.92	159.83	236.86	33.68	10.95
analyzer	196.19	409.84	195.9	482.46	25.34	15.05

Table 2: Drag and Space Savings for original inputs.

Benchmark Program	Reduced	Original	Space Saving Ratio (%)
	Reachable Integral (M $Byte^2$)	Reachable Integral (M $Byte^2$)	
javac	340.99	353.36	3.5
jack	47.92	61.39	21.94
raytrace	540.97	755.84	28.43
jess	561.68	591.09	4.98
euler	7320.18	7725.46	5.25
mc	7043.01	7513.95	6.27
juru	314.9	351.76	10.48
analyzer	859.85	1051.57	18.23

Table 3: Drag and Space Savings for alternate inputs.

4.1 Space Savings

For each of the benchmarks Figure 2 shows graphs of the the reachable heap size (sum of the sizes of the reachable objects) and the in-use heap size (sum of the sizes of the in-use objects) before and after the rewriting of the code. The black line shows the reachable size and the thin gray line the in-use size before rewriting. The area between these two lines is the initial drag. The white line shows the reachable size and the thick gray line the in-use size after rewriting. (In most cases the thick gray line and the thin gray line coincide and cannot be differentiated.) The area between the black line and the white line is the space that is saved by the optimizations.

Looking at the graphs for SPECjvm98 benchmarks, we see that the size of the reachable heap is reduced for `javac` and `jack`. Moreover, the reachable and in-use object size lines for the optimized version occur "earlier" in the graph than for the original run. This is due to the elimination of some unnecessary allocation.

For `raytrace` the size of the reachable heap is reduced by an almost constant size, and the in-use object size remains the same. This is due to the fact that close to 1MB of allocation of long-lived never-used objects has been eliminated. However, there is practically no shift in the graph since 1MB is less than 1% of the total number of bytes allocated during a run.

There is a potential for drag reduction by rewriting some of the JDK code. We demonstrate drag reduction due to JDK rewriting in `jess` benchmark. In principal, JDK rewriting is applicable for all of the benchmarks. The size of the reachable heap for `jess` is reduced by a constant size, similarly to `raytrace`.

The graph for `db` is not shown. There are no space savings for this benchmark.

Turning to the other benchmarks, for `euler` the size of the reachable heap for the original run has a constant size, because all allocations are done in advance. By assigning null to dead references we were able to reduce most of the drag (76% of it), and the optimized heap size almost coincides with the in-use object size.

In `mc` the size of the reduced reachable heap is even below the size of original in-use object size. This is due to the fact that many allocations are eliminated.

In `juru` there is a constant reduction in the reachable heap size. `juru` acts in cycles, with the same reduction on every cycle.

Lastly, for the `analyzer` benchmark the size of the reachable heap is reduced only after allocating the first 78MB in the program. This occurs because objects used for the first part of computation (first 78MB of allocation) are not needed later in the computation.

In Table 2, we show measurements of the drag reduction and the total space savings. Following Agesen *et al.* [1] we measure the space-time products for the reachable and in-use object size (we call these products integrals as they are the area under the reachable and in-use graphs respectively). The ratio between the reduced reachable integral and the original reachable integral captures the average space savings.

The original drag is the difference between original reachable and in-use integrals. The amount of drag reduction is the difference between the original reachable integral and reduced reachable integral. Drag savings is computed as the ratio between the amount of reduced drag and the original drag.

The average space savings for all the benchmarks (including db) is 14% and the average drag savings is 51%. In mc the size of the reduced reachable heap is less than the size of the original in-use objects. This leads to 168% savings of drag, since we saved even more than the original drag.

We also ran each benchmark on an input other than the one initially analyzed by the tool. Results are shown in Table 3. For raytrace, euler, mc, juru and analyzer space saving results were similar to the ones reported for the initial input. For javac, jack and jess some space is saved, although less than the amount of space saved for the initial input. We believe that this shows that the transformations work for multiple inputs, noting that the approximation of the amount of space savings in general should be based on measurements for a set of inputs.

We informed the developers of juru and analyzer of our results. These developers will be integrating our suggested rewritings into future versions of their code. Interestingly, we also noted that later versions of jack application, which is now called javacc [11], use similar rewritings to the ones we suggest.

4.2 Runtime Savings

The measurements were done on a 400MHz Pentium-II CPU with 128MB main memory running Windows NT 4.0 on several JVMs including IBM JVM 1.3 [10] and Sun HotSpot Client 1.3 [14]. For the SPECjvm98 benchmarks, juru and analyzer we used a 32MB initial heap size and a 48MB maximum heap size. For euler and mc we used a 64MB initial heap size and a 96MB maximum heap size. Each reported result is the average of 10 runs.

Table 4 shows the runtime savings for the optimized benchmarks. We choose to show runtime results for Sun HotSpot client since it uses a generational GC [28]. A generational GC delays the collection of some unreachable objects in order to get better performance. Thus, the potential benefit for saving drag time for an object is decreased. Nevertheless, the average runtime for all of the benchmarks (including db) is reduced by 1.07%.

Speedups are due to two factors:(i) allocation savings, saving both allocation and constructor time and (ii) GC is invoked less frequently.

Benchmark Program	Sun HotSpot 1.3 Client		
	Reduced Runtime (sec.)	Original Runtime (sec.)	Runtime Saving (%)
javac	26.569	26.538	-0.12
jack	14.961	15.11	0.99
raytrace	16.07	16.452	2.32
jess	11.868	12.116	2.05
euler	42.772	43.605	1.91
mc	34.154	34.884	2.09
juru	23.51	23.69	0.76
analyzer	15.958	15.898	-0.38

Table 4: Runtime Savings.

5. STATIC ANALYSIS ALGORITHMS

In this section, we discuss the kind of program analysis algorithms necessary to automate our space savings program transformations. In the future, we hope to implement these algorithms. Our goal is to find analyses that are cheap yet provide sufficient information to do the transformations.

In our discussion we provide examples chosen from the manual transformations we did on the benchmarks. Table 5 shows the transformations used for each benchmark, the space saved for each transformation, and the static analysis technique that might be used to achieve the savings automatically.

We believe that *demand* analysis [6, 22] may be useful in this context. This would allow starting the analysis from allocation sites with large drag as obtained by our tool.

There are many potential program analyses that could be useful and our discussion is by no means complete. Specifically we consider the following program analysis issues:

- Dataflow analyses (Section 5.1). We consider coarse-grained analysis that provides sufficient information.
- Analyzing references versus analyzing arrays of references, which is considerably more complex (Section 5.2).
- The code granularity of analysis (Section 5.3). Here, we tried to pick the smallest unit of code that can be analyzed separately.
- Dependencies on call graphs (Section 5.4). The presence of virtual function invocations and uncalled methods greatly affects the results; call graph information shows the methods that may actually be used.
- Exception analysis (Section 5.5). Java's precise exception model [7, Chapter 11] forbids certain program transformations.

We also make certain simplifying assumptions about the analyzed code, which are satisfied by the benchmarks we considered. We assume that all the code for an application is available, and it does not include reflection or multithreading (which can be handled conservatively by methods not described here).

5.1 The Dataflow Analysis Problem

We found four kinds of useful dataflow information:

Usage-analysis Finding variables that are set using side-effect free expressions, but never used. This helps to find assignment statements that can be safely eliminated.

Indirect-usage analysis The main idea is that an object is never-used if none of its references is ever dereferenced.

Liveness-analysis Identifying program locations where a reference has no future use, i.e., it is set before being

Benchmark Program	Rewriting Strategy	Reference Kinds	Drag Saving (%)	Expected Analysis
`javac`	code removal	protected	21.8	indirect-usage
`jack`	lazy allocation	package	70.34	min. code insertion
`raytrace`	code removal	private array	45.01	array liveness (R)
	assigning null	private	6.27	liveness (R)
`jess`	assigning null	private array	2.7	array liveness
	code removal (JDK rewrite)	public static final	1.68	usage
	code removal	private static	11.09	usage (R)
`euler`	assigning null	package array	76.46	array liveness
`mc`	code removal	local variable + private	119.95	indirect-usage (R)
	assigning null	private array	48.87	array liveness
`juru`	assigning null	local variable	33.68	liveness
`analyzer`	assigning null	local variable + private static	25.34	liveness

Table 5: Summary of Rewritings.

used on every execution path. This information can be passed to GC, as done in Agesen *et al.* [1], so that the root set is reduced at runtime. Alternatively, the program can be transformed to assign null to dead references.

Minimal-code insertion This analysis helps to determine where lazy allocation could be used. It is similar in spirit to code motion, e.g., see [4].

Two examples of the benefits of usage-analysis are shown in Table 5. We describe one of them here. In the locale class of the JDK, a static variable is declared for every possible locale. These variables are assigned with newly allocated locale objects. The code of a benchmark (along with JDK classes) can be analyzed for usage of these static variables and those, which are never-used, can be eliminated. This saves the allocation of the corresponding locale object.

Two examples of the benefits of indirect usage-analysis are shown in Table 5. We describe one of them here. In a class in `javac` a string is allocated and assigned to an instance field. The field is never used except for assigning its value to other reference variables. These variables are never used; thus, the allocation of the string can be saved.

We see that in `analyzer`, liveness analysis for local variables together with liveness analysis for a private static variable lead to a 25% reduction in drag.

Minimal code insertion is employed for lazy allocation. It is used to approximate the possible first use of an object. At first, possible references to that object are identified using alias analysis [5, 9]. Then, possible uses of a reference are identified using use-def chains [18]. Finally, the code for lazy allocating the object is inserted before every possible use. Minimal code insertion is achieved by analyzing the places where such code is inserted in a PRE [17, 4] fashion. In general this type of transformation could lead to longer running times for the programs; thus, it must be applied judiciously. We employed it for `jack` with good results; runtime results show an improvement of close to 1%.

5.2 Simple References vs. Arrays of References

An array of references represents a set of references, whereas a reference variable represents a single reference. Moreover, the address of an array element also depends on a subscript variable that is computed at execution time. Thus, analysis for array variables, e.g., liveness, is harder than analysis for simple variables. In `jess` a dynamic vector-like array of references is maintained. After removing the logically last element from this array, that element has no future use. Interestingly, the original code tries to handle this case of a dead element, but it does not handle it completely. Array liveness analysis [24] can detect this case. This stresses the need for an automatic solution.

5.3 Analysis Granularity

The cheapest analyses handle small code units, e.g., Agesen *et al.* [1] perform liveness analysis one method at a time. This analysis would suffice to reduce the drag in `juru` by 34%. However, there are cases in other benchmarks, e.g., for liveness of an instance field, where whole program interprocedural analysis would be required since non-local information must be considered.

We are also investigating variants of interprocedural analysis limited to a smaller code unit, e.g., a set of classes, but do not report on it here.

5.4 Dependencies on Call Graphs

Considering the call graph can reduce the amount of code that needs to be analyzed, thus leading to more accurate results. In particular, the call graph shows the methods that are never called (unreachable methods) and can be used to reduce the set of possible targets for a virtual call site. In Table 5, we mark usage of call graph dependencies by *(R)*. In `raytrace` there is an instance field of a class that is not used outside of the constructor, except for a get method that returns the value of the field. The call graph shows that the get method is never invoked.

5.5 Dependencies on Exception Analysis

The precise exception model of Java requires careful analysis in order to enable the movement of code or the removal of code. Our transformations involve code removal, thus the removed code must be analyzed for the exceptions that it can throw. Then, the rest of the code must be analyzed to verify that none of these exceptions could be caught by an exception handler.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a useful profiling tool for investigating heap memory behavior of Java programs and saving space. Our experiments indicated that it is quite easy to use the tool in order to reduce the space without a deep understanding of the application.

We have also identified ways in which optimizing compilers could automatically achieve most of these benefits. In the future we hope to develop feasible compiler algorithms that can achieve part of these savings.

7. REFERENCES

[1] O. Agesen, D. Detlefs, and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279, June 1998.

[2] J. M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, 1977.

[3] B. Blanchet. Escape analysis for object oriented languages. application to Java^{tm}. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, Denver, 1998.

[4] R. Bodík, R. Gupta, and M. L. Soffa. Complete removal of redundant expressions. *ACM SIGPLAN Notices*, 33(5):1–14, May 1998.

[5] A. Deutsch. Semantic models and abstract interpretation for inductive data structures and pointers. In *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'95*, pages 226–228, New York, NY, June 1995. ACM Press.

[6] E. Duesterwald, R. Gupta, and M. Soffa. A practical framework for demand driven interprocedural data flow analysis. In *Trans. on Prog. Lang. and Syst.*, 1998.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesely, 1996.

[8] G. W. Hamilton. Compile-time garbage collection for lazy functional languages. In *Memory Management, International Workshop IWMM 95*, 1995.

[9] M. Hind and A. Pioli. Which pointer analysis should I use? In *Int. Symp. on Soft. Test. and Anal.*, 2000.

[10] IBM JDK 1.3. Available at http://www.ibm.com/java.

[11] JavaCC - The Java Parser Generator. Available at http://www.metamata.com/javacc.

[12] Java Grande Benchmark Suite. Available at http://www.epcc.ed.ac.uk/javagrande.

[13] Sun JDK 1.2. Available at http://java.sun.com/j2se.

[14] Sun HotSpot Client 1.3. Available at http://java.sun.com/products/hotspot.

[15] T. P. Jensen and T. Mogensen. A backward analysis for compile-time garbage collection. In *European Symp. on Prog.*, pages 227–239, 1990.

[16] R. Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1999.

[17] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[18] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[19] W. D. Pauw and G. Sevitski. Visualizing reference patterns for solving memory leaks in java. In *ECOOP'99*, pages 116–134, Lisbon, Portugal, 1999.

[20] S. Porat, B. Mendelson, and I. Shapira. Sharpening global static analysis to cope with java. In *CASCON*, 1998.

[21] N. Röjemo and C. Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN Int. Conf. on Func. Prog.*, pages 34–41, Philadelphia, Pennsylvania, 24–26 May 1996.

[22] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comp. Sci.*, 167:131–170, 1996.

[23] M. Serrano and H.-J. Boehm. Understanding memory allocation of scheme programs. In *International Conference on Functional Programming*, pages 245–256, 2000.

[24] R. Shaham, E. K. Kolodner, and M. Sagiv. Automatic removal of array memory leaks in java. In *Int. Conf. on Comp. Construct.* Springer, Apr. 2000.

[25] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in java. In *Int. Symp. on Memory Management.* ACM, Oct. 2000.

[26] SPECjvm98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[27] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Conf. on Object-Oriented Prog. Syst., Lang. and Appl.*, 1998.

[28] P. R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management, International Workshop IWMM*, Sept. 1992.