HUP: A Heap Usage Profiling Tool for Java Programs

Michael Pan*
Supervisors Dr. Elliot Kolodner and Dr. Mooly Sagiv
School of Computer Science, Tel-Aviv University, Israel

September 1, 2001

Abstract

This thesis presents a Heap Usage Profiling tool (HUP) for exploring and reducing heap space consumption in Java applications. The space saving is based on the fact that some of the allocated objects are not immediately used (or not used at all) in the application code. Also, there are objects, which though no longer in use, remain reachable and in memory. The HUP tool allows a programmer to locate and remove memory bottlenecks, which are caused by unused objects. The usefulness of the HUP tool is demonstrated by applying it to several complex applications that use heap space heavily.

*pan@post.tau.ac.il

Acknowledgments

I would like to thank Dr. Mooly Sagiv for his guidance, help and support throughout this work. I would like to thank Dr. Elliot Kolodner for his advice, ideas and help throughout this work.

This work was done in collaboration with Ran Shaham and I would like to thank him for his advice, ideas and help. Thanks to Roman Manevich for reading drafts and for his helpful comments. Thanks also to Leonid Bobovich, Michael Rozhavsky and Gleb Natapov.

I would like to thank the Academy of Science, Israel and IBM (through a Faculty Partnership Award) for their financial support.

I would like to thank my wife Diana for her great patience.

Contents

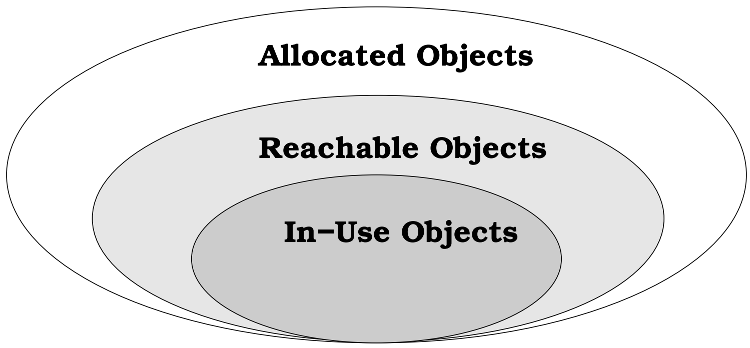

Figure 1: Reachable vs. in-use heap objects.

1 Introduction

Memory is one of the most critical resources in many applications. The program heap, in turn, is the space that is aggressively used during program execution, especially in object-oriented languages. Unwise heap usage may make the program infeasible or degrade performance. Furthermore, programs that employ garbage collection require larger heaps. Motivated by this our research is focused on reducing the heap space in the presence of a Garbage Collector (GC), particularly in a Java environment. Generally, a Java program allocates objects and the GC algorithm is responsible for collecting the objects, which are no longer in use and reclaiming their space. However, commonly used GC algorithms do not collect all potential garbage but only those objects that are no longer reachable from the *root set*, where the root set is the set of references, which are stored in global variables or on the stack. Yet, there are objects that are reachable from the root set at a given point in the program and will not be used in the future. This is illustrated pictorially in Figure 1.

Some of the unused but reachable objects may be reclaimed in order to save space. Moreover, on some occasions, we can delay the allocation of used objects, and thereby reduce the heap consumption. These ideas motivate the classification of the lifecycle of an object as shown in Figure 2. We refer to the time interval from the allocation time of an object until it is first used as *lag time* and to the object itself as a *lagged object*. The time interval from the last use of an object until it becomes unreachable is called *drag time* and object itself is said to be a *dragged object*. In a special case, when the object has no uses at all, we refer to the interval between its allocation and the point it becomes unreachable as *void time* and the object itself as a *void object* (Figure 3). These definitions of lag, drag and void were first introduced by Röjemo and Runciman [RR96].

Previous work by Shaham, Kolodner and Sagiv [SKS00] showed a potential space saving for SPECjvm98 benchmarks ranging from 23% to 74% in dragged and void objects. Moreover, in [SKS01], they applied simple code

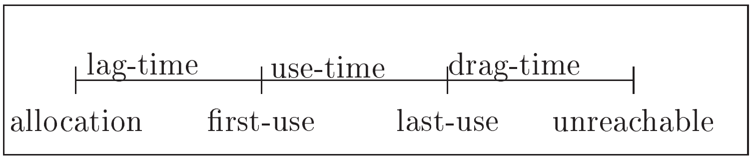

Figure 2: The lifetime of used objects.

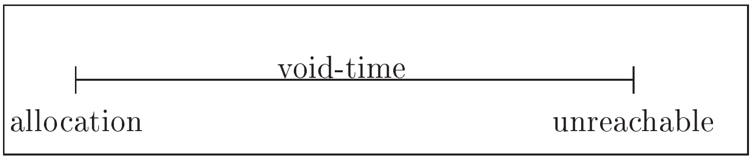

Figure 3: The lifetime of void objects.

transformations to these benchmarks and saved on average 18% of the space. These results were obtained by a simple profiling tool prototype, which instrumented Sun's Java Virtual Machine (JVM) [SUN01]. In this research we concentrate on the profiling approach. We propose a flexible and powerful tool for profiling heap usage measurements and examine its usefulness for complex Java applications.

1.1 Main results

The contributions of this thesis can be summarized as follows:

- We developed a new tool for the runtime profiling of Java programs called **H**eap **U**sage **P**rofiler (HUP), which is a non-trivial task, and involves several state of the art techniques.
- The tool offers many features for profiling heap usage beyond the prototype reported in [SKS01], including measuring and analyzing lagged objects, which allows more space savings.
- The tool is built on top of JVMPI [VL00], which is a generic JVM [LY96] profiling interface. The tool works on any JVM implementation, which supports JVMPI. In particular, for a specific application the tool can be applied on several different Java Virtual Machine implementations, in order to better tune the performance of a given application.
- In order to demonstrate the usefulness of HUP, we applied it to several complex applications written in Java that use heap space heavily. The

results are reported in Section 3. We are encouraged by the fact that in the Soot application [VRHS+99] we were able to save on average 17% of the maximum heap size using HUP, even though we were not familiar with this code at all. These improvents have already been integrated into Soot by the code owners. We also used HUP to locate a space bottleneck in TVLA [LAS00], leading to an average improvement of 15% in the maximum size of the heap. Since TVLA is heap intensive, this allows it to be applied to larger problems than previously feasible.

1.2 Design goals

During design of the HUP tool we tried to achieve the following goals:

- *Functionality.* The purpose is the collection of object usage information during the execution of a Java application and analysis of the collected data. We would like the HUP tool to be conservative and record all object uses wherever they come from, Java or native methods code. The HUP profiling mechanism is required to maintain the behavior of a profiled application. The profiled information should be presented in such a way, that the user would be able to understand and analyze the profiling results.

- *Usability.* The HUP tool is required to be easy to use. We would like the HUP tool to require minimum interaction with the user for application profiling and provide a user-friendly interface for exploring profiling results.

- *Portability.* We would like HUP to be platform and JVM independent, i.e., to work for all JVM implementations and on all platforms.

- *Extendability.* The HUP design should allow adding of new features into its code.

- *Performance.* The runtime overhead of the HUP tool during application profiling should be as small as possible. Feasibility for large applications is an additional related requirement.

The profiling data collection *functionality* and the *portability* requirements are achieved by using of the JVMPI interface and bytecode instrumentation. The JVMPI interface allows collection of different runtime information directly from a JVM implementation. Unfortunately, the JVMPI does not provide all the information we require. To overcome this problem, we use bytecode instrumentation, which changes the application source code in such a way that the required information will be accessable through the JVMPI. The bytecode instrumentation that we propose does not change the application results. We discuss the combination of the JVMPI interface and the bytecode instrumentation in Section 4.2 and the complete description of the bytecode instrumentation is given in Section 5.

The profiling data analysis *functionality* is achieved by the HUP-analysis tool that provides a user-friendly interface in order to explore the profiling results. HUP-analysis allows a user to make various queries on profiled data and locate memory bottlenecks in a profiled application. The HUP-analysis facilities are discussed in Section 2.2.

The HUP tool is *usable* since an application profiling procedure is simple and requires almost no interaction with the user. This result is mainly achieved by an "online" instrumentation of an application class files, which we discuss in depth in Section 5.3.

The design and development of the HUP tool were made with *extendability* requirement in mind. The combination of the JVMPI and the bytecode instrumentation can be extended in order to collect more profiling information in the future. The HUP-analysis code allows adding new queries on profiled data, we discuss some of them in Section 8.2.

The current version of the HUP tool is not tuned for high *performance* and creates significant runtime overhead during application profiling. We discuss the HUP performance problems and propose solutions for them in Section 8.1.

1.3 Outline of the rest of this thesis

The rest of this thesis is as follow. Section 2 presents an overview of the HUP tool architecture and the HUP-analysis facilities. Section 3 presents the results of applying the HUP tool to complex Java application. Section 4 discusses important design decisions, the definition of object usage and the integration of the JVMPI interface into the HUP architecture. Section 5 gives a detailed description of the instrumentation techniques we used. Section 6 describes the profiling agent, which is HUP's core engine for profiling data collection. Section 7 presentes related work. Section 8 concludes the thesis and describes the current limitations of HUP and proposals for further research.

2 System Description

2.1 Architecture

The HUP tool was designed to work independently with any JVM implementation, which support the JVMPI interface. Currently, we have tested it on IBM JDK 1.3 [IBM01] and on Sun JDK 1.3 [SUN01] classic. During execution, it uses the BCEL [Dah99] bytecode instrumentation framework, written in Java and freely available on ``http://bcel.sourceforge.net''. The tool was developed and used on the Windows 2000 operating system. Most of the code is platform independent. The platform dependent code includes some C-code that deals with sockets and memory management, and batch files for running HUP. The tool is distributed under GNU General Public License (see Section A.4).

The HUP architecture is shown in Figure 4. An application written in the Java language is compiled by a Java compiler into corresponding class files. Each class file contains a binary representation of one application class or interface; in particular, the class file contains the bytecode of the class. The class files are read and executed by a JVM.

The execution of any application by a JVM requires the JDK runtime classes that comprise the Java platform's core API. `InstrumentJDK` is a program that receives as input all the JDK classes and outputs the corresponding instrumented JDK classes together with some additional information. This program is invoked once for any given JDK and then the results are utilized when an application program is profiled. The instrumented JDK classes are passed to the JVM and replace the standard bootstrap and other JDK classes during the execution of an application. The JDK class information contains a list of the JDK classes and a description of native methods that are found in these classes. This information is passed to the profiling agent. The profiling agent communicates with the JVM implementation and collects data from the execution of a given application. Then, the collected data is processed by the HUP-analysis. The HUP-analysis, in turn, provides a user interface for querying information about the profiled data such as information about lag, drag and void objects.

2.2 Analysis

First, we introduce some definitions, which are used in the following discussion. We measure the time as bytes allocated since the beginning of program execution. This provides a less-machine dependent measure of time. Observing the size of reachable objects as function of time, we calculate the

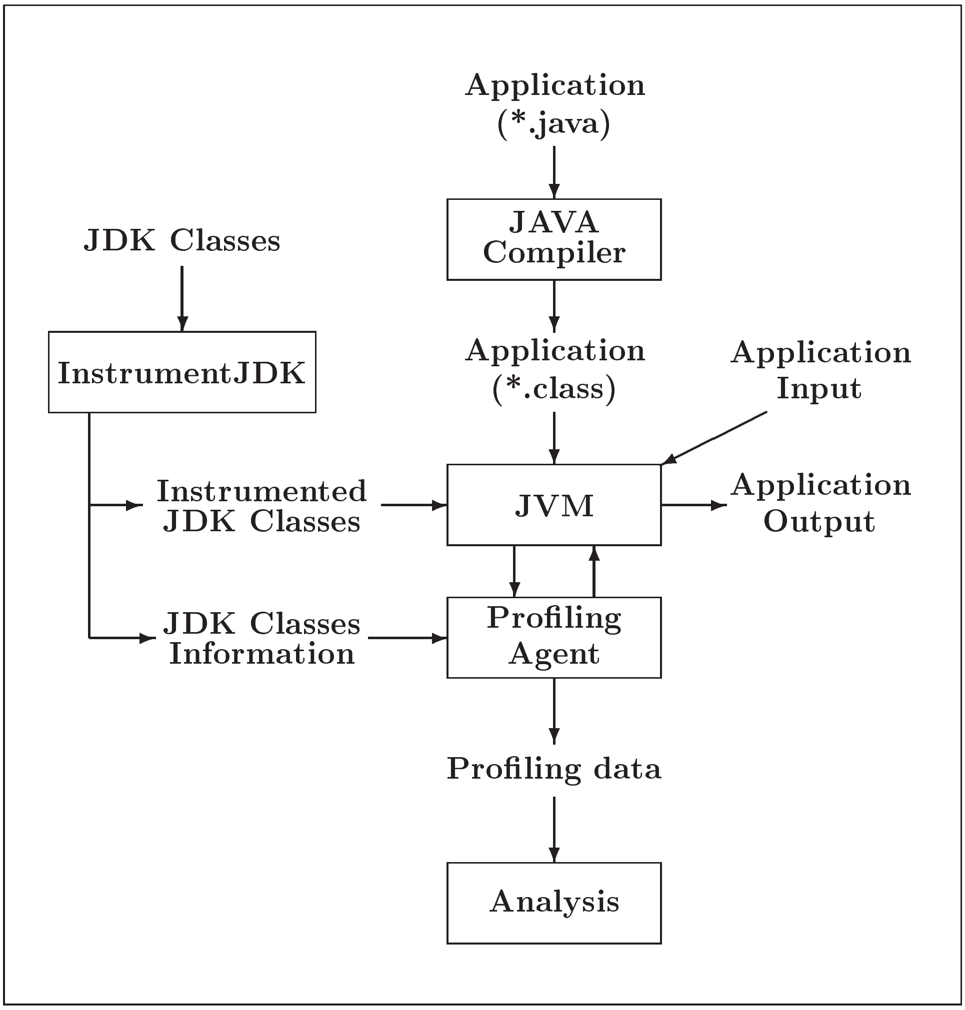

Figure 4: A scheme of the HUP architecture.

integral of this function. We refer to this space-time integral value as the *total space* of a given application. Similarly, we refer to the values of the integral of the lag, drag and void size functions as *total lag, drag and void space* respectively. The HUP-analysis calculates an approximation for these integrals as follows. The product of object lifetime and object size defines the *object lifetime space* for a single object and the sum of the lifetime space of all objects gives the application's *total space*. Similarly, the product of an object's lag, drag or void time and its size defines the *object's lag, drag and void space* respectively. The sum of *lag, drag* or *void space* of all objects produce *total lag, drag and void space* of the application respectively. The total lag, drag and void space definitions are particularly useful for understanding the impact of the lagged, dragged and void objects on the heap consumption of a given application.

The HUP-analysis is based on the classification of lagged, dragged and void objects. Specifically, the analysis classifies the objects by class, allocation site and nested allocation site. The *nested allocation-site* of an object is the call chain of methods leading to the object's allocation. In other words it is the thread stack trace at the point the object is allocated. Calling a method from a different lines of code of the same method generate different nested allocation-sites. The *allocation site* of an object is its nested allocation site of depth one, or simply the method, where the object is allocated. As opposed to the nested allocation site definition, the allocation site definition does not distinguish between different lines within the same method. The analysis calculates lag, drag and void space for a given class, by summing the lag, drag and void space of all instances of this class, respectively. In the same way, the analysis calculates lag, drag and void space for allocation sites and nested allocation sites. The HUP-analysis displays items in each of the above classifications sorted by lag, drag or void space, thereby bringing user attention to the items with the most significant lag, drag and void space.

In general, the analysis process is as follows. First, the user selects a class which appears to have a significant lag, drag or void space. Then, the user queries for the allocation sites of the selected class. In case further investigations are required, the user may query for nested allocation sites of all or some allocation sites that were previously obtained. The stack traces in the HUP-analysis usually contain the Java source file name and the line number in the file, so the user can easily locate an allocation site and a nested allocation site in source code.

Another classification of objects that is provided by HUP-analysis is based on the differentiation of objects by their lifetime patterns. The lifetime pattern is defined by the stack traces of the first and the last object use. For example, when the user examines the lagged objects of a given nested allocation site, objects with different stack traces of first uses could be identified. This difference may point to different roles of objects in a given program run, even though they are allocated at the same point. In the same way, the user may explore the lifetime patterns of dragged objects.

A complete description of the HUP-analysis user interface is found in the User's Manual (appendix A).

3 Applications

In this section, we report the results obtained by applying the HUP tool to the SPECjvm98 benchmarks, Soot and TVLA applications. For each of these applications we provide a brief description and the lag, drag and void space statistics that were collected for it. Then, we present the improvements we made in the application, based on the analysis using HUP, and discuss the impact of these improvements.

In order to reduce heap consumption of a given application, we manually rewrote the application code. Our code rewriting reduced the application lag, drag and void space by using the following approaches:

- Decreasing of lag, drag and void time of lagged, dragged and void objects.
- Reducing the number of lagged, dragged and void objects.
- Reducing the size of lagged, dragged and void objects.

In following discussion, we propose several techniques for code rewriting and show examples of applying these techniques to the applications.

Changing the program code, we preserved the correctness of the program. We applied source transformations only after a thorough inspection of the source code and validation that the original and revised programs produce identical results on several inputs. Moreover, the proposed changes in Soot and TVLA code were verified and accepted by developers of these applications.

3.1 SPECjvm98 benchmarks

SPECjvm98 [SPE98] is a standard benchmark suite for the performance comparison of JVM implementations. We employed six benchmarks from SPECjvm98, which use significant amounts of heap memory. The profiling results for these benchmarks are shown in Figure 5. The graphs in Figure 5 shows the applications' lag, drag, void and in-use space, where the X axis is time in bytes and the Y axis is the heap size.

We choose one representative of the benchmarks, `raytrace`, for analysis. The `raytrace` benchmark has significant lag, drag and void space. Specifically, 27.57% of the total space is drag, 7.76% is lag and 31.52% is void.

3.1.1 Program transformations

In order to reduce the heap consumption of `raytrace` we applied the following techniques.

- *Dead code removal.* Dead code does not affect the result of the program. There may be local variables, instance variables and array elements that reference objects, which are never used. The *dead code removal* technique eliminates the allocations of these objects, thereby reducing the number of void objects. In order to apply this technique, we must ensure that the constructor and the finalizer of a given object have no influence on the rest of the program, e.g., they do not update other objects or static variables. An additional check for the constructor is that it does not throw an exception for which there may be a handler in the surrounding code.

  Another application of the *dead code removal* technique is the removal of the unused object's fields from an appropriate class declaration. The removal of the unused fields reduces the class instance size, thereby reducing the space consumption of a given application. In some cases, there are assignments to the object's fields, but the assigned values are never obtained from the fields. These object's fields can be removed after the removal of the assignments to these fields.

- *Lazy allocation.* This technique changes the code to allocate objects lazily. In particular, it eliminates the original allocation of the object and the variable that would have referred to the object remains null or is assigned null. The object's allocation is moved to its first use by inserting code before each possible first use to check that the reference variable is still null and if so, to allocate the object. We refer to this technique as *lazy allocation* and it is particularly useful for lagged and void objects, because it decreases lag time of lagged objects and eliminates allocations of void objects. In order to apply this technique, we must ensure that the code in the object constructor does not depend on program state, e.g., the values of its parameters are independent of the execution path leading to the constructor. Also, we must ensure

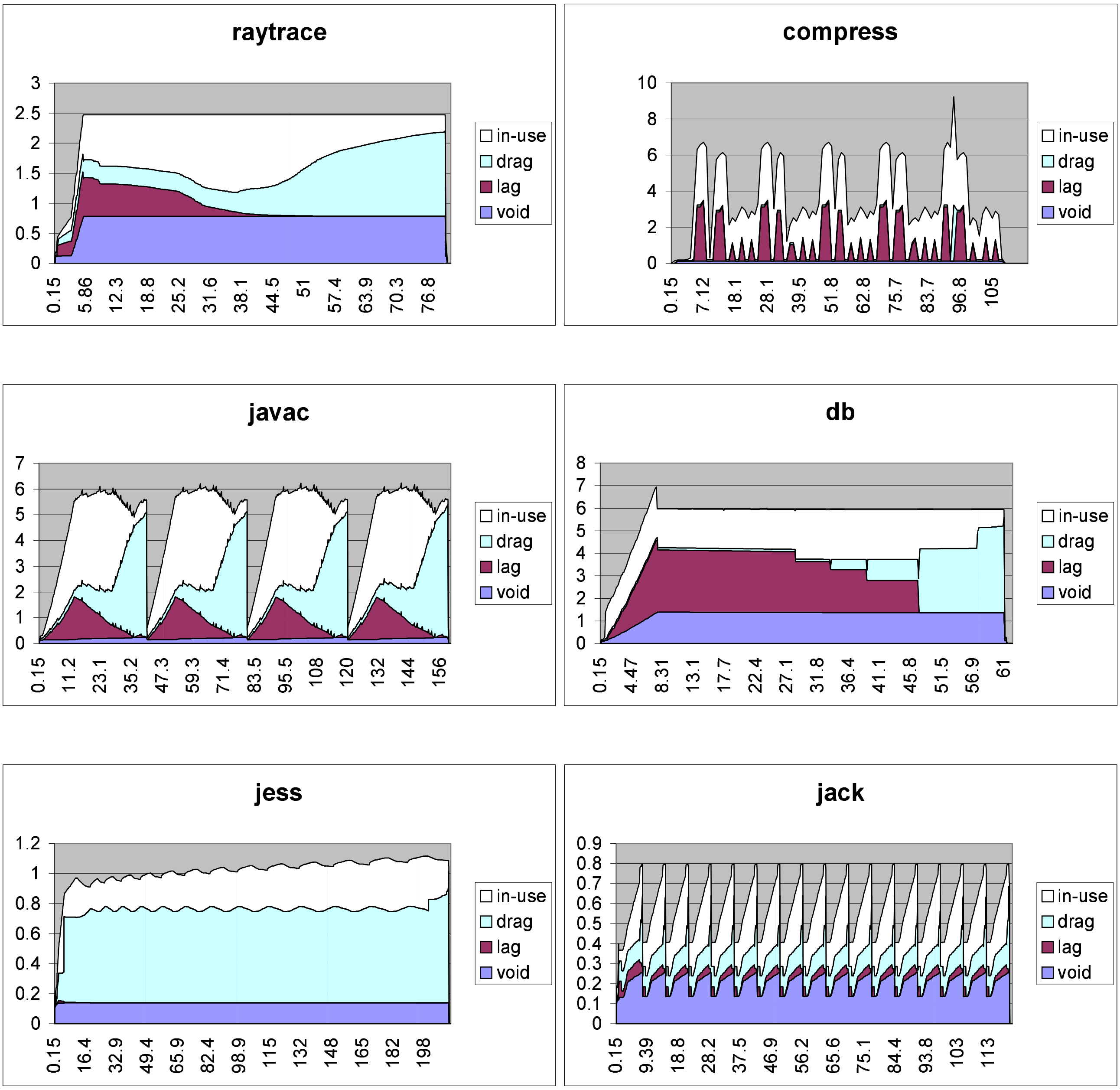

Figure 5: SPECjvm98 benchmarks profiling results. X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

that the code in the constructor and finalizer does not influence the rest of the program. Lazy allocation adds runtime overhead for the checks at every possible first use and requires extra synchronization in multithreaded applications, which may slow the execution.

- *Assigning null.* There are objects that remain reachable after their last use. In case there are no possible uses of an object after some point in program, the reference that keeps the object reachable is called a *dead reference.* Assigning a null to a variable with dead reference allows the GC implementation to free the object space. This technigue reduces heap consumption by decreasing the drag time of objects.

- *Array length reduction.* If the object is an array and there are array items that are never used, we can allocate an array of a smaller size. In this case we must change indexes for all array accesses in the application code in such a way that it would not affect the result of the program. This technique reduces the size of lagged, dragged and void objects.

3.1.2 Raytrace

The HUP tool was applied to `raytrace`. We analyzed the results with HUP-analysis and then improved the `raytrace` code by applying aforementioned techniques.

We applied the *dead code removal* technique to the objects, which we found to be allocated and assigned to array elements, but never used. Specifically, the HUP analysis indicated that the objects of class `Point` produce the most significant void space. The investigation of the allocation sites of this class, using HUP analysis, indicated that almost all of void objects of class `Point` are allocated in method `CreateFaces` of class `OctNode`. Examination of the code of this method showed that it allocates objects and assigns them to array elements. Nested allocation sites that correspond to the method, indicated lines in which there are allocations of objects that are never used. Further examination of the code led to the conclusion that these unused objects are assigned to array elements, which are never accessed in the rest of the program code. We verified in the code that the `Point` object constructor does not affect the rest of the program, and then applied *dead code removal* by removing the allocations of these objects. This transformation reduced close to 75% of the total void space in `raytrace`.

The *lazy allocation* technique was applied to objects, which were not found to be used immediately after their allocation. For example, observing allocation sites of lagged objects, in HUP analysis, we found that the method

`Initialize` in class `OctNode` contributes 20% to the total lag space. Corresponding nested allocation sites indicated that this method allocates arrays of objects, which are not immediately used. We observed the code of the method and found that it is called from the `OctNode` constructors and the allocated arrays are assigned to the private variables of the class instance. Array objects have no constructor in Java, so we could safely apply the *lazy allocation* technique. The HUP analysis indicated that the same allocation sites that produce the lagged array objects in the `Initialize` method, also contribute around 2% to the total void space. Thus, in addition to the lag space that the *lazy allocation* reduced, it also reduced the `raytrace` void space.

In the same way, we applied the lazy allocation technique to the `CacheIntersectPt` objects allocated in the class `CacheIntersectPt`, to the `Point` objects allocated in the class `IntersectPt` and to array of integers allocated in the class `Canvas`. These allocation sites were found by examination of the classes of lagged objects in the HUP-analysis and saved an additional 12% of the total `raytrace` lag space.

The *array length reduction* technique was applied to dragged arrays with unused items. We were not able to reduce the drag time of these arrays directly, but we noticed that we could reduce their size. Specifically, the HUP analysis indicated that arrays of objects contribute the most to the drag space. We examined the allocation sites of this class, using HUP analysis, and found that 16% of the total drag space consist of arrays of objects that are allocated in the constructor of class `Face`. Investigation of the code showed that class `Face` allocates an array of `Point` objects of length 4 in its constructor. This array is referenced by a private variable and the class provides two public methods for obtaining items from the array and for the updating of its elements. These methods receive as input the index of the array item. We looked for the uses of these methods in the `raytrace` code and found that the method that obtained the array elements is called only for the first and last indexes of the array. We reduced the array length to 2 and added code to the array access methods, which changes the input indexes in such a way that the array length change is transparent to the rest of the program.

We applied the *assigning null* technique to dragged arrays. These arrays were allocated and used in `PolyTypeObj` constructors and then remained referenced by the private variables in the constructed objects. Using HUP analysis we found the nested allocation site of these arrays of objects, which contribute more than 4% of the total drag space. Observing the lifetime patterns corresponding to this allocation site we found that 99% of the drag space at that allocation site was produced in patterns corresponding to a last use in the methods of the `TriangleObj` class. The remaining 1% of the drag space at that allocation site was produced in patterns corresponding to a last use in the methods of the `PolygonObj` class. We observed the code of these classes and found the following. Classes `TriangleObj` and `PolygonObj` extend the abstract class `PolyTypeObj`, which receives in its constructor an array of `Point` objects. This array is used in `PolyTypeObj` constructor for calculations and then its reference is stored in a `PolyTypeObj` private variable. The `PolyTypeObj` constructor is explicitly invoked in `TriangleObj` and `PolygonObj` constructors. The `PolygonObj` uses this array during the run, while `TriangleObj` does not. In order to reduce the drag space, we decided not to store this array for objects of `TriangleObj` class and used *assigning null* technique to free this array. In particular, we change the variable that references the array to be protected and in the constructor of `TriangleObj`, after the invocation of `PolyTypeObj` constructor, the variable is set to null, thereby allowing GC to free the array. This transformation reduced almost 4% of drag space.

After applying of above transformations we ran HUP on the modified `raytrace` benchmark and obtained the following results. Space saving in total lag is 39%, in total drag is 22% and in total void is 77%. Space saving in total space is 38% and total allocations are reduced by 6%. There is a 32% reduction in maximum heap size and no change in execution time. The execution time measurements were made on the Sun JVM implementation [SUN01] with the default runtime settings. Figure 6 shows the reduction results.

3.2 Soot

Soot is a Java optimization framework. It can be used as a stand-alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode. Soot aggressively uses the heap during its run. For example, using Soot to optimize the SPECjvm98 benchmark suite, which involves analysis and optimization of 32 class files, requires at least 43Mb heap size. Figure 7 shows the profiling results for Soot optimizing SPECjvm98 benchmark suite.

In order to reduce heap consumption in Soot we applied the *abstract data type (ADT) changing* approach. The ADT changing approach proposes to change the program internal data structures and algorithms in such a way, that the new program implementation will use less space. The implementation of this approach varies from program to program and usually requires deep knowledge of the program's algorithms, data structures and features of its input. ADT changing must be done very carefully; it can affect per-

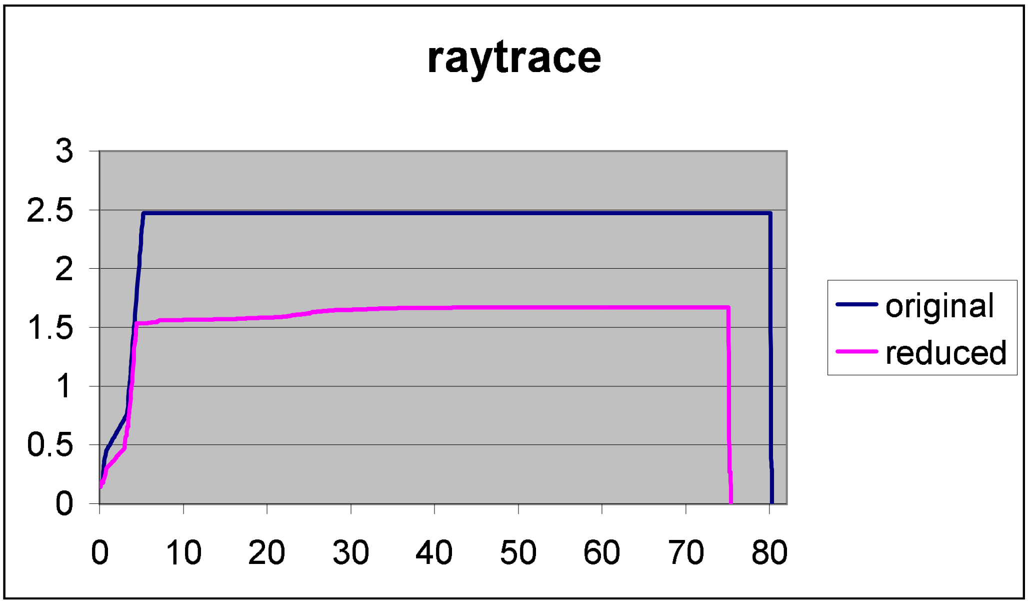

Figure 6: Raytrace space reduction results. X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

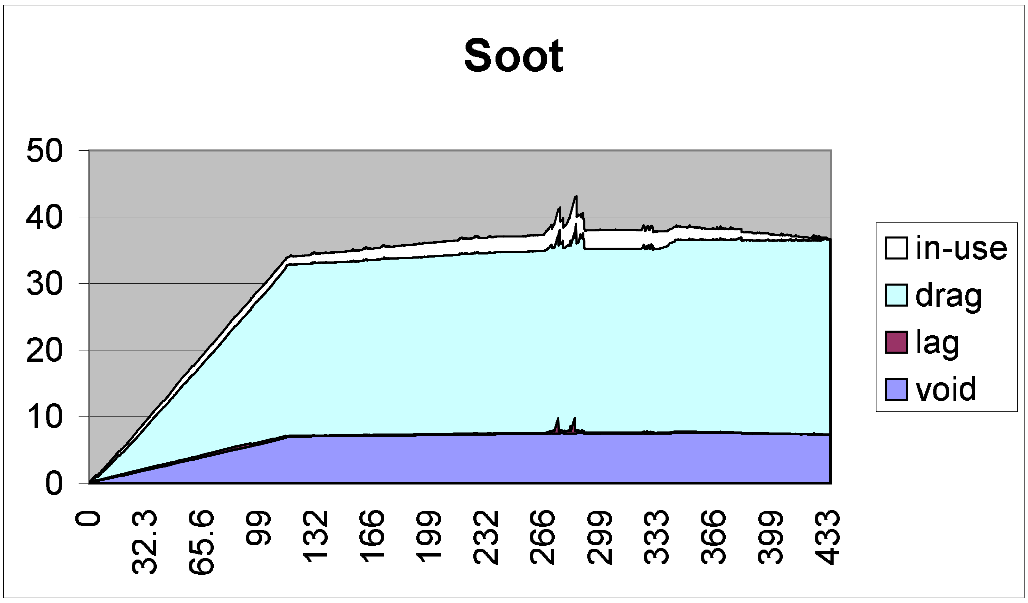

Figure 7: Soot profiling results on SPECjvm98 benchmark suite. X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

formance and involve complex changes, which themselves can influence heap consumption.

Soot loads all classes of the program being optimized into memory and then performs optimization. We found that Soot creates an inefficient in-memory representation of the classes. During the loading of the classes, it repeatedly creates objects, which represent the same data and stores them in memory. We applied *ADT changing* by collecting these objects in a global hash table and allowing them to be shared, thereby avoiding the creation and storing of similar objects. Notice that this transformation reduces the total drag by reducing the number of dragged objects. A detailed description of this transformation follows.

Observing nested allocation sites of lagged and dragged objects in HUP-analysis we found, that arrays of characters, which are allocated in the `readConstantPool` method of the class `ClassFile` contribute more than 6% to drag space and 12% to void space. Examination of the Soot code showed that this array is filled with data from the constant pool of the class file that Soot receives as input. The constant pool is the section of the class file, which holds all of the symbolic data needed by the class. This data includes symbolic references to fields, classes, interfaces, and methods used internally by this class, as well as important symbols such as the name of the class and the names of its fields and methods. The array of characters, which was found to be problematic, is used by Soot in the `CONSTANT_Utf8_info` class, which represents the `CONSTANT_Utf8` records in the class file constant pool. `CONSTANT_Utf8` records frequently appear in the constant pool and contain information such as class, method and field names, method signatures and constant strings. We queried the HUP-analysis and found that there are a lot of dragged instances of `CONSTANT_Utf8_info` class. Therefore, we concluded that there are `CONSTANT_Utf8` records in class files, which are not used by Soot for its optimizations.

As noted earlier, `CONSTANT_Utf8` fields of a given class file are represented in `CONSTANT_Utf8_info` objects. These objects are stored in a list, which is referenced by the object that represents the class file. We assumed that the class files being analyzed could contain a lot of similar `CONSTANT_Utf8` records, for example, methods names and their signatures, which comprise the symbolic references of methods being invoked in the code of the classes. We stored the `CONSTANT_Utf8_info` objects in a global hash table, thereby avoiding the creation of `CONSTANT_Utf8_info` objects with the same content. It leads to a savings between 5% to 10% of the maximum heap size on various inputs.

The ADT changing approach was also used in the code of the `SootField` and `SootMethod` classes and reduced on average 5% of the maximum heap.

	HelloWorld	**SPECjvm98**	**TVLA**
Number of input files	1	32	160
Maximum heap saving	20%	17%	18%
Total space saving	12%	20%	not available
Execution time	unchanged	unchanged	unchanged

Table 1: Soot space reduction results

An additional 5% of the maximum heap was saved by applying the lazy allocation technique in the `SootMethod` class code. All of these changes were accepted by Soot developers and integrated into the Soot code.

Soot was tested on three different inputs. The first input was a simple HelloWorld program with only one class file, the second input was the SPECjvm98 benchmark suite, which involved the analysis of 32 class files, and the third input was the TVLA program, which involved the analysis of 160 class files. The tests were run with the HUP tool, except for the TVLA input, which is too large to be run with HUP. Results for the TVLA input, were obtained by running the JVM with the `-verbose:gc` option. Tests results are shown in Table 1 and Figure 8 shows reduction results for the SPECjvm98 input. The execution time measurements were made on the Sun JVM implementation [SUN01] with the default runtime settings.

3.3 TVLA

TVLA (Three-Valued-Logic Analysis engine) is a framework for automatically constructing static-analysis algorithms from an operational semantics, where the operational semantics is specified using logical formulae. Space saving for the TVLA application is especially important because the TVLA program usually requires huge amounts of space for its run. For example, in one of the tests the maximum heap of size exceeded 300MB. Figure 9 shows the profiling results for TVLA execution, which performs correctness checking of the bubble sort algorithm.

The work on reduction of memory usage of the TVLA program was done in cooperation with one of the TVLA developers, who verified the usefulness and usability of the HUP tool.

TVLA uses an iterative algorithm and during each of the iterations creates massive number of objects, which remain in memory after the end of the iteration. These objects are not used any longer and are usually replaced by new objects in the following iterations. Some of these objects are not replaced and remain in memory till the end of the TVLA execution. We

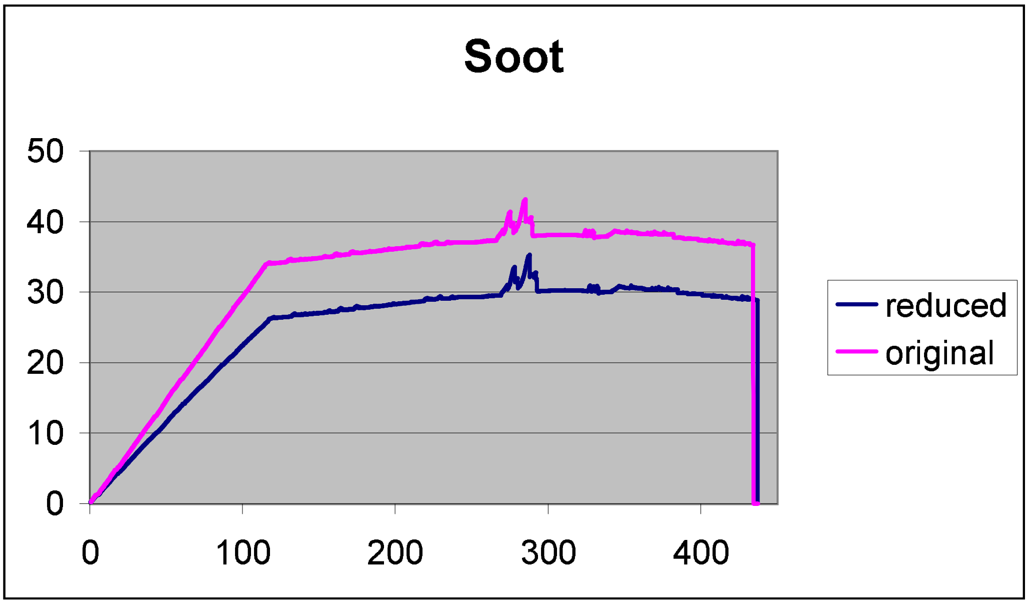

Figure 8: Soot reduction results on SPECjvm98 benchmark suite. X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

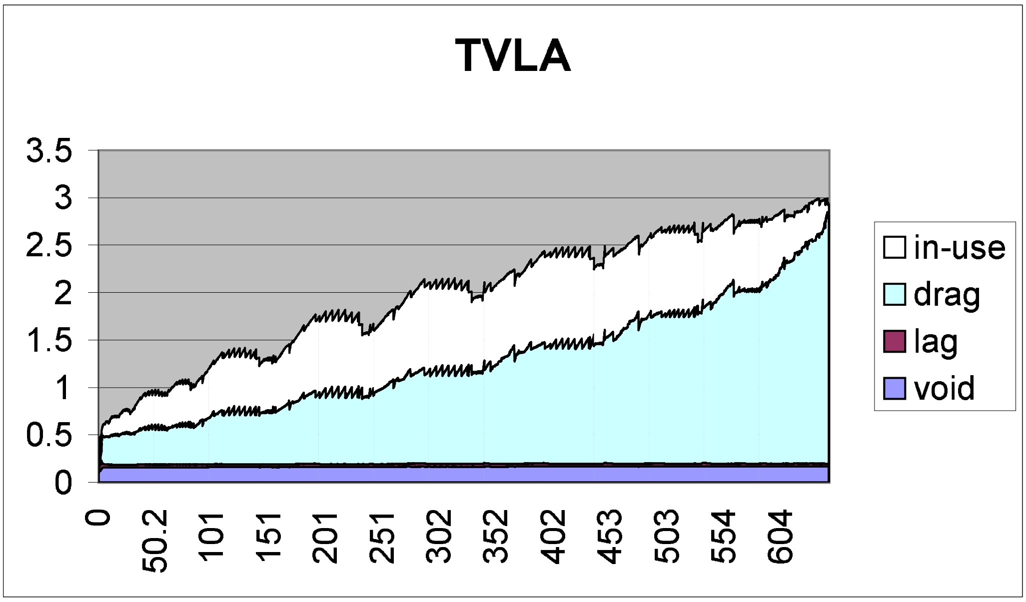

Figure 9: TVLA profiling results (correctness checking of the bubble sorting algorithm). X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

applied the *assigning null* technique to these objects, thereby allowing the GC implementation to free these objects at the end of the algorithm iteration. A detailed description of this transformation follows.

During exploration of the TVLA profiling results, the HUP-analysis indicated that allocations in the constructor, `put` and `rehash` methods of class `HashMap` contribute more than 25% to drag space. Observing the nested allocation sites that correspond to these `HashMap` methods in HUP-analysis, showed that most of the calls to these methods are made in the `blur` method of class `NaiveBlur`. Examination of the TVLA code showed that there are objects of class `HashMap`, which are created and updated in the `blur` method. Then references to these `HashMap` objects are stored in `NaiveStructure` objects. We queried the HUP-analysis and found that the `HashMap` objects allocated in the `blur` method create significant drag space. Finally, we observed the code of constructor, `put` and `rehash` methods of class `HashMap` and concluded that the objects allocated in these methods are dragged because their references are stored in the dragged `HashMap` objects.

In order to apply the specified static analysis algorithm, TVLA builds a *control flow graph* (CFG) of a given input program and applies an iterative data-flow algorithm by updating the information stored in nodes of the CFG. The part of the runtime information that is stored in the CFG nodes are the `NaiveStructure` objects, which in turn contain the dragged `HashMap` objects. Each time the algorithm starts updating the information stored in a CFG node, it usually creates new versions of `HashMap` objects for this node. After a CFG node is processed, the node's `HashMap` objects remain reachable from it and are not freed. Next time (if ever) the algorithm updates this CFG node, the `HashMap` objects are replaced with new ones, and can be freed by GC implementation. In order to reduce heap consumption in TVLA, we applied the *assigning null* technique at the end of each iteration of a data-flow algorithm and explicitly assigned null to the `NaiveStructure` object fields, which hold references to the dragged `HashMap` objects.

After applying the above transformation we ran TVLA (correctness checking of the bubble sorting algorithm) with HUP and found a 9% savings in total space. Figure 10 shows the results for this run. We performed tests of TVLA with another static-analysis algorithm that performs pointer analysis for the merging of two sorted lists. In these tests we verified that the space reduction holds for multiple static-analysis algorithms, which are implemented in the TVLA framework. Tests results are shown in Table 2. The execution time measurements were made on the Sun JVM implementation [SUN01] with maximum heap size of 400 megabytes.

	correctness checking	pointer analysis
Maximum heap saving	10%	21%
Total space saving	9%	not available
Execution time improvement	2.5%	1.5%

Table 2: TVLA space reduction results

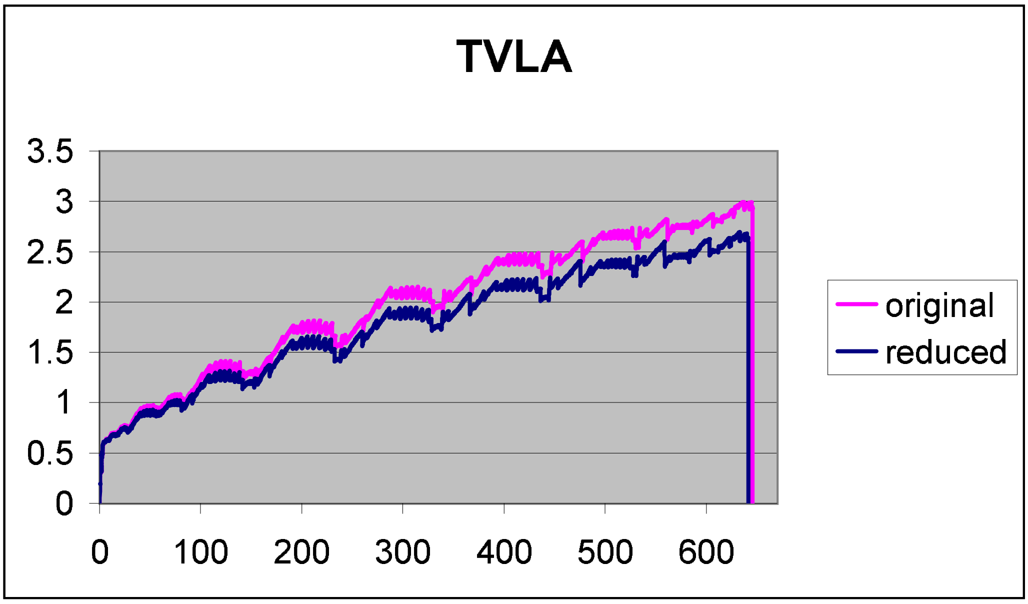

Figure 10: TVLA space reduction results (correctness checking of the bubble sorting algorithm). X-axis denotes allocation time in MB. Y-axis denotes heap size in MB.

4 HUP Rationale

4.1 Object usage definition

In order to precisely define lagged, dragged and void objects, we need to clarify what does it mean for an object to "be used". Choosing the right definition of object usage is extremely important because it directly affects the definition of lag, drag and void. For example, previous work [RR96, SKS01] conservatively assume that whenever an object is accessed, by means of either assigning a new value or retrieving the object's value (e.g., used as an actual parameter), it is potentially used. As we shall see, this definition leads to empty lag space in Java programs, even though, the allocation of some objects can be delayed. Motivated by this, we decided to look for a new object usage definition.

We consider several approaches for the definition of object usage.

1. An object is *used* if its content is accessed in expression e in the program.

2. An object is *used* if its content is accessed in expression e in the program, excluding the code of the constructor.

3. An object is *used* if its content is accessed in the R-value of an expression e in the program, excluding the constructor invocation.

In the above definitions we refer to data that is stored in an object as object content. Object content may contain the object's fields, monitor and class information. Read or write operations on an object's content are considered as accessing of object content.

Figure 11 contains code fragments to demonstrate the differences between the above definitions. The first defintion counts object usage in the assignment `int j = a.i` in Figure 11(a). Indeed, the expression `a.i` accesses content of the object. As we shall see, the first definition also counts object usage for the new operation `A a = new A()`. The second definition differs from the first one in fact, that it does not count the accessing of object's content inside the object constructor. In Figure 11(b), second definition determines the first object usage in the assignment `int j = a.i`, although, the constructor is not empty and contains code, which is counted as object usage by the first definition. The third definition counts the first object usage in Figure 11(c) in the assigment `int j = a.i`. In order to understand the difference between this definition and the previous ones, consider the assignment `a.i = 0`. The object's content is accessed in `a.i`, but it is found in the

```
class A {
   public int i;
   public A() {}
   public A(int j) { this.i = j; }
}


(a)

   A a = new A();
   int j = a.i;

(b)

   A a = new A(1);
   int j = a.i;

(c)

   A a = new A(1);
   a.i = 0;
   int j = a.i;
```

Figure 11: An example of code fragments demonstrating different object usage definitions.

```
(a)

   Object obj;
   obj = new Object();


(b)

   NEW (Class java.lang.Object)
   DUP
   INVOKESPECIAL (Method java.lang.Object())
   ASTORE (local variable index)
```

Figure 12: Java object allocation example.

L-value of the expression, thus the third defnition does not count object usage for it, whereas the two others do. In the following discussion we describe how each of the above definitions affect the definitions of lag, drag and void objects and choose the most appropriate definition for object usage.

The first definition leads to an almost empty lag space. The reason is that object usage immediately follows object creation. Figure 12(a) shows typical Java code that creates a new object and Figure 12(b) shows appropriate bytecode that is produced for it by the Java compiler. The `INVOKESPECIAL` operation invokes the object constructor and retrieves object content to resolve the constructor method. Thus, the definition will count non-empty lag space only for array objects, which have no constructors in Java.

The next two definitions are more interesting. The second definition proposes to exclude object content accesses that occur during execution of the constructor. Thus, the first object usage is the first access to object content after the constructor completes. This is illustrated in Figure 13. In the rest of this thesis we refer to this definition as the *constructor excluding approach*.

The constructor excluding approach is simple and comprehensible, but assumes that the program is written in a good programming style, where constructors do only simple initialization operations. Unfortunately, object initialization code can be extremely complex. Moreover constructors may contain code that is part of the program algorithm and not part of object initialization.

The third approach defines object usage by the accessing of object content

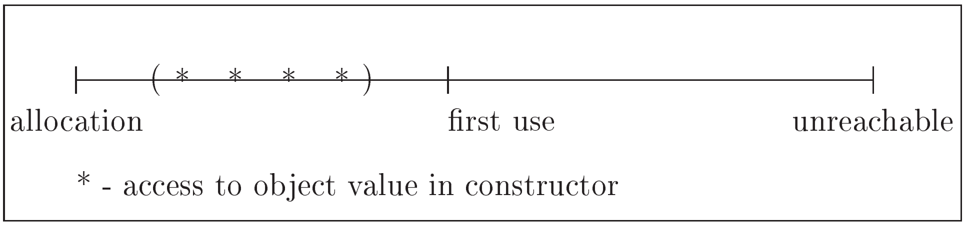

Figure 13: Object usage in constructor excluding approach.

in the R-value of an expression. Such an access of an object in an R-value of an expression involves read operations on object content. We refer to such operations as *get operations*. Similary, accessing object's content in an L-value of a given expression involves write operations on object content and we refer to such operations as *put operations*. Table 3 and Table 4 divide the Java bytecodes into put and get operation groups. In order to count object uses, according to the third definition, we simply need to count the *get* group operations on the object during application execution. In the rest of this thesis we refer to this definition as the *put/get approach*.

Method invocation opcodes are placed into the *get* operation group, because they read data from object space during method resolution. As we mentioned above, the object constructor invocation is done by an `INVOKESPECIAL` operation and leads to an almost empty lag space. To overcome the problem we exclude the invocation of the object constructor from object usage.

The *put/get* approach is conservative for the definition of drag. Indeed, *put* operations, which are not followed by any *get* operation can be conservatively discarded.

There are two problems with the *put/get* approach. First, it may falsely assume an object usage, even that it can be easily delayed. For example, code in Figure 14(a) contains a *get* operation, but its execution can be delayed. Second, the execution of a *put* operation may depend on the application state or on the state of the application environment. Thus, *put* operations may define an object usage, which the *put/get* approach would not discover. For example, the statement in Figure 14(b) performs a *put* operation on `object_1` assigning the value of `object_2.field` to the `object_1.field`. This *put* operation can not be moved into a later place in the code because the value of `object_2.field` will no longer be the same.

The *put/get* approach seems to have several advantages over the *constructor excluding* approach. It defines object usage without any relation to where it is placed in code, inside or outside the constructor. This is particularly important in the Java language, where array objects do not have

Opcode	Description
CHECKCAST	Check whether object is of given type.
INSTANCEOF	Determine if object is of given type.
ATHROW	Throw exception or error.
MONITORENTER	Enter monitor for object.
MONITOREXIT	Exit monitor for object.
GETFIELD	Fetch field from object.
AALOAD	Load *reference* from array.
BALOAD	Load *byte* or *boolean* from array.
CALOAD	Load *char* from array.
DALOAD	Load *double* from array.
FALOAD	Load *float* from array.
IALOAD	Load *int* from array.
LALOAD	Load *long* from array.
SALOAD	Load *short* from array.
ARRAYLENGTH	Get length of array.
INVOKEINTERFACE	Invoke interface method.
INVOKESPECIAL	Invoke instance method; special handling for superclass, private, and constructor methods.
INVOKEVIRTUAL	Invoke instance method.

Table 3: Java *get* operations.

Opcode	Description
PUTFIELD	Set field in object.
AASTORE	Store into *reference* array.
BASTORE	Store into *byte* or *boolean* array.
CASTORE	Store into *char* array.
DASTORE	Store into *double* array.
FASTORE	Store into *float* array.
IASTORE	Store into *int* array.
LASTORE	Store into *long* array.
SASTORE	Store into *short* array.

Table 4: Java *put* operations.

```
(a)
   object.field_1 = 0;
   object.field_2 = object.field_1

(b)
   object_1.field = object_2.field;
   object_2.field++;
```

Figure 14: Examples for *put/get* approach problems.

constructors. Finally, it simplifies the analysis of lagged objects, because it bounds the operations that need to be examined as candidates to be delayed (only *put* operations need to be considered).

In order to understand the difference between the *put/get* approach and the *constructor excluding* approach, we conducted runtime experiments. For this purpose, we implemented them in HUP and measured them on the SPECjvm98 benchmarks. We report the results in Table 5. The second column presents the difference between the lag space that was counted by the *put/get* definition and lag space that was counted by the *constructor excluding* definition. The third and fourth columns present comparison of lag space that was counted by *put/get* and *constructor excluding* definitions with the total application space. In five out of the six tested benchmarks, the HUP tool found significant lag space. The benchmark with no significant lag - `Jess` is less important for comparison, because it does not produce enough lagged objects of any definition to understand the relationship between the definitions. On the other hand, we can see that both of the definitions counted similarly small lag in this benchmark.

Between the benchmarks with significant lag, there is only one benchmark - `Javac`, which showed a significant difference between the definitions of lag (20%). The rest of the benchmarks showed almost no difference, less than 1%. Due to the small differences between the lag measured according to the definitions and the reasons we mentioned above, we decided to adapt the *put/get* approach for the object usage definition.

Benchmark	Difference in Put/Get and Constructor Excluding	Put/Get Lag Space vs. Total Space	Constructor Excluding Lag Space vs. Total Space
Compress	0.00%	24.79%	24.79%
Db	0.17%	29.76%	29.81%
Jack	0.08%	5.01%	5.00%
Javac	20.02%	13.77%	17.21%
Raytrace	0.67%	7.76%	7.63%
Jess	19.82%	0.12%	0.15%

Table 5: Lag measurements results for *put/get* approach.

4.2 The usage of JVMPI

There are two main approaches to allow Java profiling of programs in a JVM implementation independent way. In the first approach, the information is collected directly from the JVM, requiring a standard profiling interface to be supported in the JVM. JVMPI [VL00] is such an interface, which is intended to become a standard and to be supported by all JVM implementations. The general structure of JVMPI is shown in Figure 15. The JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent. The virtual machine notifies the profiler agent of various events, for example, those corresponding to heap allocation, thread start, etc. The profiler agent issues controls and requests for more information through the JVMPI. For example, the profiler agent can control which event notifications are reported to it.

Using JVMPI has several advantages. JVMPI provides easy access to runtime information, which is difficult to obtain in other ways. For example, it allows dumping heap topology and threads stack traces. The JVMPI is gaining acceptance by the developer community as a basis for constructing Java applications profilers [BG01]. Establishing JVMPI as a standard ensure that HUP would work with any JVM implementation supporting it. Furthermore, little or no change will be needed in HUP, when a new version of a JVM appears. Finally, the JVMPI performance may be improved in future JVMs, thereby improving the performance of the HUP. Unfortunately, JVMPI does not provide, all the information we require.

The second approach, source to source, is based on changing the original program source code. By definition, the changed code will be portable across JVM implementations. But this approach is problematic. First, pro-

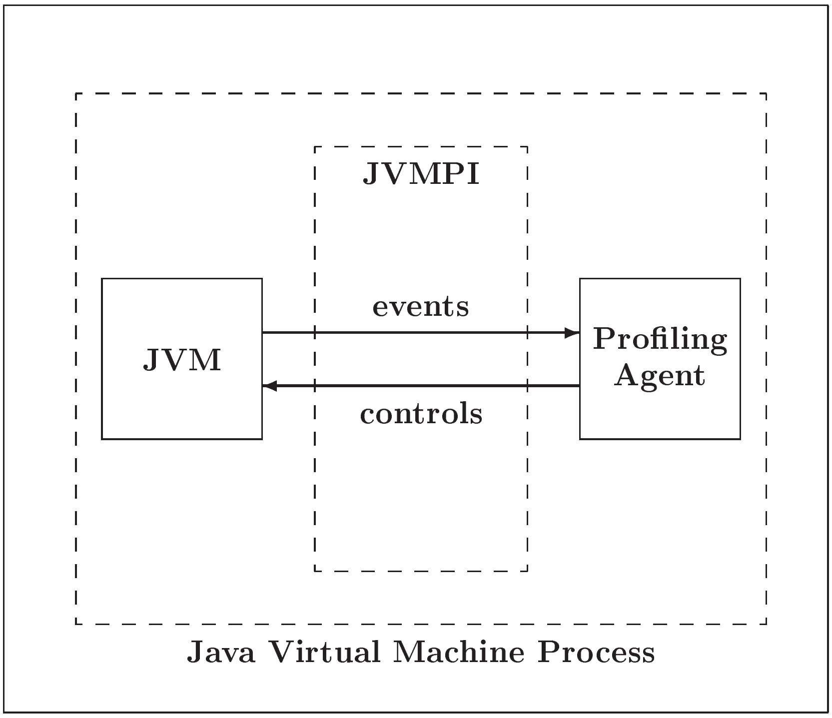

Figure 15: JVMPI.

filing code that is injected into the program may significantly slow down the program execution. Second, changing the program source code may influence the program behavior. In particular, the creation of objects for profiling may change heap consumption. For example, the simple string printing operation, `System.out.print(new String("HUP"))` generates a new `String` object. In this case, the profiler must provide a mechanism that excludes such objects from the output data.

HUP uses a combination of the JVM profiling interface and source to source approaches to collect profiling information. Our profiling agent runs in the scope of the JVM and obtains relevant information from JVMPI events. In order to obtain the information, which is not available through JVMPI, we use the source to source approach, but make sure that it does not change heap consumption. For example, for the object uses, like `getfield`, which are not reported by the JVMPI, we apply the following technique.

- A new public method is added to `java.lang.Object` class. All Java classes inherit from `java.lang.Object` and thus inherit this new method.
- A call to this new method is injected in places the `getfield` or another unreported operation appears.

Every method invocation triggers the `JVMPI_EVENT_METHOD_ENTRY2` event, which reports to the profiling agent the object that was accessed. Certainly, the method invocation does not affect heap consumption. We refer to this technique as the *generating usage event* (GUE) technique and it is discussed in depth in Section 5.

5 Instrumentation

5.1 GUE technique

As we mentioned in Section 4, HUP uses JVMPI to collect profiling information. JVMPI does not provide events for every object usage. Actually, the only object usage operation that JVMPI reports is the method invocation event `JVMPI_EVENT_METHOD_ENTRY2`. This event reports the method name and the object on which it is invoked. The list of unreported bytecode operations is shown in Table 6. The *put* operations are not counted as object uses, but can help in object usage analysis. Particulary, the HUP tool collects information about the first and the last *put* operations on object, which are useful for object lifetime pattern analysis.

Get operations	Put operations
AALOAD	AASTORE
BALOAD	BASTORE
CALOAD	CASTORE
DALOAD	DASTORE
FALOAD	FASTORE
IALOAD	IASTORE
LALOAD	LASTORE
SALOAD	SASTORE
GETFIELD	PUTFIELD
MONITORENTER	
MONITOREXIT	
CHECKCAST	
INSTANCEOF	
ATHROW	

Table 6: Unreported operations.

```
public void __hup__GetOperation() {}
public void __hup__PutOperation() {}
```

Figure 16: New methods in `java.lang.Object`

In order to generate JVMPI events for unreported operations we apply bytecode instrumentation. To instrument the bytecode we use the BCEL bytecode instrumentation framework. In the first phase of the instrumentation we add to the `java.lang.Object` class two new methods to enable us to report *put* and *get* operations separately, shown in Figure 16.

All the Java classes inherit from `java.lang.Object` class, thus all Java classes inherit these new methods. In order to generate JVMPI events for unreported operations, we add calls to the `__hup__GetOperation` or `__hup__PutOperation` method, dependent on the operation type. The call is inserted before or after the operation; the order is not important due to the insignificant execution time of the operation. Injected-calls trigger `JVMPI_EVENT_METHOD_ENTRY2` events and report object usage to the profiling agent. Recalling from Section 4, we refer to this technique as the generating usage event (GUE) technique.

The methods we add to `java.lang.Object` in the GUE technique are empty. In general, the JVM can automatically remove calls to the empty methods in order to speed up the execution. We assume, that the JVM implementation is obliged to report through JVMPI the method calls, which are present in application code, even if these calls are not performed by the JVM due to its optimizations. This assumption holds for the JVM implementations, which are currently supported by the HUP tool.

5.2 Object uses inside native methods

Counting accesses to objects from native methods is a non-trivial task. We distinguish between JDK and application native methods. Java applications use the *Java Native Interface (JNI)* [AP00] to access Java objects from inside the native methods. The JNI interface is a set of C-functions. Pointers to these functions are stored in the `JNIEnv` structure, which is passed as an argument to native methods. Object usage operations through the JNI are reported by the JVMPI. However, some operations are not reported by the JVMPI. A list of unreported JNI operations is shown in Table 7.

Get Functions	**Put Functions**
GetObjectField	SetObjectField
GetBooleanField	SetBooleanField
GetByteField	SetByteField
GetCharField	SetCharField
GetShortField	SetShortField
GetIntField	SetIntField
GetLongField	SetLongField
GetFloatField	SetFloatField
GetDoubleField	SetDoubleField
GetBooleanArrayRegion	SetBooleanArrayRegion
GetByteArrayRegion	SetByteArrayRegion
GetCharArrayRegion	SetCharArrayRegion
GetShortArrayRegion	SetShortArrayRegion
GetIntArrayRegion	SetIntArrayRegion
GetLongArrayRegion	SetLongArrayRegion
GetFloatArrayRegion	SetFloatArrayRegion
GetDoubleArrayRegion	SetDoubleArrayRegion
GetObjectArrayElement	SetObjectArrayElement
GetBooleanArrayElements	
GetByteArrayElements	

Table 7: JNI unreported functions.

Get Functions	Put Functions
GetCharArrayElements GetShortArrayElements GetIntArrayElements GetLongArrayElements GetFloatArrayElements GetDoubleArrayElements GetArrayLength	
ReleaseBooleanArrayElements ReleaseByteArrayElements ReleaseCharArrayElements ReleaseShortArrayElements ReleaseIntArrayElements ReleaseLongArrayElements ReleaseFloatArrayElements ReleaseDoubleArrayElements	
MonitorEnter MonitorExit	
IsInstanceOf GetObjectClass	
Throw	
GetStringLength GetStringChars ReleaseStringChars GetStringRegion	
GetStringUTFLength GetStringUTFChars ReleaseStringUTFChars GetStringUTFRegion	
GetPrimitiveArrayCritical ReleasePrimitiveArrayCritical GetStringCritical ReleaseStringCritical	

Table 7: JNI unreported functions.

For the operations, which are not reported by the JVMPI, we apply JNI instrumentation. Specifically, the profiling agent changes the content of the JNIEnv structure and replaces pointers to unreported functions by the agent's wrapper functions. The wrapper functions use the GUE technique to report object usage and then call the original JNI function. The profiling agent has an initialization handler, JVM_OnLoad, which is called during JVM initialization. The initialization handler gets as input a pointer to the JNIEnv structure. We assume that there is only one copy of JNIEnv structure in the JVM and the profiling agent gets a pointer to this copy. The profiling agent changes the content of JNIEnv, replacing references to the original JNI functions with the instrumented wrapper functions.

A special case is JDK native methods. JDK native methods may use the knowledge of internal JVM structures and make non-conventional access to objects (not through the JNI). An example of a non-conventional access is the System.arraycopy method, which uses memcpy to speed the execution of copying an array. There is a possibility that the JVMPI events would not be fired on such operations. We assume that a JDK native method accesses only its arguments in a non-conventional way and does not ”travel” through the heap to access other objects. Further, we conservatively assume that JDK native methods access all its arguments in a non-conventional way. In order to report the usage of the JDK native method arguments to the profiling agent, we use the GUE technique. Usage of arguments can be reported before or after the call to native method. The order is not important because JDK native methods make few allocations.

Usage of arguments is reported by a wrapper method. We create a wrapper method for each JDK native method and insert the wrapper method into the class file, which contains the original native method. A wrapper method has the same signature as the original native method, but a different name, __hup__<original-name>. A wrapper method generates usage events for its arguments of reference type and, thus, reports usage of the arguments. Then it calls the original JDK native method.

Once the wrapper methods are added to the JDK classes, we need to replace all the calls to the JDK native methods by calls to the wrapper methods. However, this is not sufficient. The reason is that there can be a method in the JDK or in application classes, which has the same name and signature as one of JDK natives. At the time we replace the call to the JDK native method by the call to our wrapper, we do not know which class's method will be called during the execution. For example, Figure 17(a) shows abstract class A, which has two subclasses B and C. For simplicity, suppose that all of them are JDK classes. Class B implements method METHOD as native and class C implements the same method in Java. Figure 17(b) shows the situation after the wrapper method was added to class C. In this situation replacing the call of METHOD on an object of type A by a call to __hup__METHOD is not safe. At runtime the object could be of type C, the call would not be resolved and Java would throw NoSuchMethodException. To overcome this problem, we rename all methods and interfaces, whose name and signature is identical with some JDK native method, with the corresponding wrapper method name. The result of this transformation is shown in Figure 17(c). Now it is safe to replace calls to the JDK native methods by calls to the wrapper methods.

In addition to calls to JDK native methods from bytecode, there are calls to these methods from native code. The HUP agent replaces the pointer to `GetMethodID` function in JNIEnv by a pointer to agent's wrapper function. Each time `GetMethodID` is called for a method, which has the name and signature of one of the JDK native methods, the agent replaces the method name by the appropriate wrapper name, thereby redirecting the call to its wrapper.

In addition to non-conventional accesses to objects from the JDK native methods, there could be such accesses inside the JVM implementation itself. For example, the JVM could store some internal information in an object and access it whenever it needs to. There is no interface in JVM, which allows collection of information about these accesses. Moreover, this information is useless for application code rewriting, because the application has no control over these accesses. We assume that the accesses to objects inside the JVM are rare and can be neglected.

5.3 Static and dynamic instrumentation

In order to replace the JDK native methods calls in a specific class file, we need to know the names and signatures of all JDK native methods. The profiling agent also requires this information to redirect JDK natives calls, performed through the JNI interface, to their wrappers. Motivated by this, we divide instrumentation into two phases: *static* and *dynamic*. The flow of instrumentation data is shown in Figure 18.

The static instrumentation (`InstrumentJDK`) runs offline. It collects information about native methods in JDK classes and then instruments the JDK classes. The static instrumentation outputs instrumented JDK classes, a list of JDK classes and JDK native method information (names and signatures). The instrumentation of JDK classes is done at this stage for performance reasons; it reduces the instrumentation overhead at runtime.

Dynamic instrumentation occurs at runtime; it instruments classes of the application being executed. In order to allow dynamic instrumentation, we provide an instrumentation server, which is written in Java and uses the BCEL bytecode instrumentation framework. The instrumentation server gets Java classes through a TCP socket, instruments them and sends them back to the recipient. The profiling agent, in turn, gets classes from the JVM

```
(a)
    class A
        abstract METHOD
    class B extends A
        native METHOD
    class C extends A
        METHOD
(b)
    class A
        abstract METHOD
    class B extends A
        native __hup__METHOD
    class C extends A
        METHOD

(c)
    class A
        abstract __hup__METHOD
    class B extends A
        native __hup__METHOD
    class C extends A
        __hup__METHOD
```

Figure 17: JDK natives instrumentation

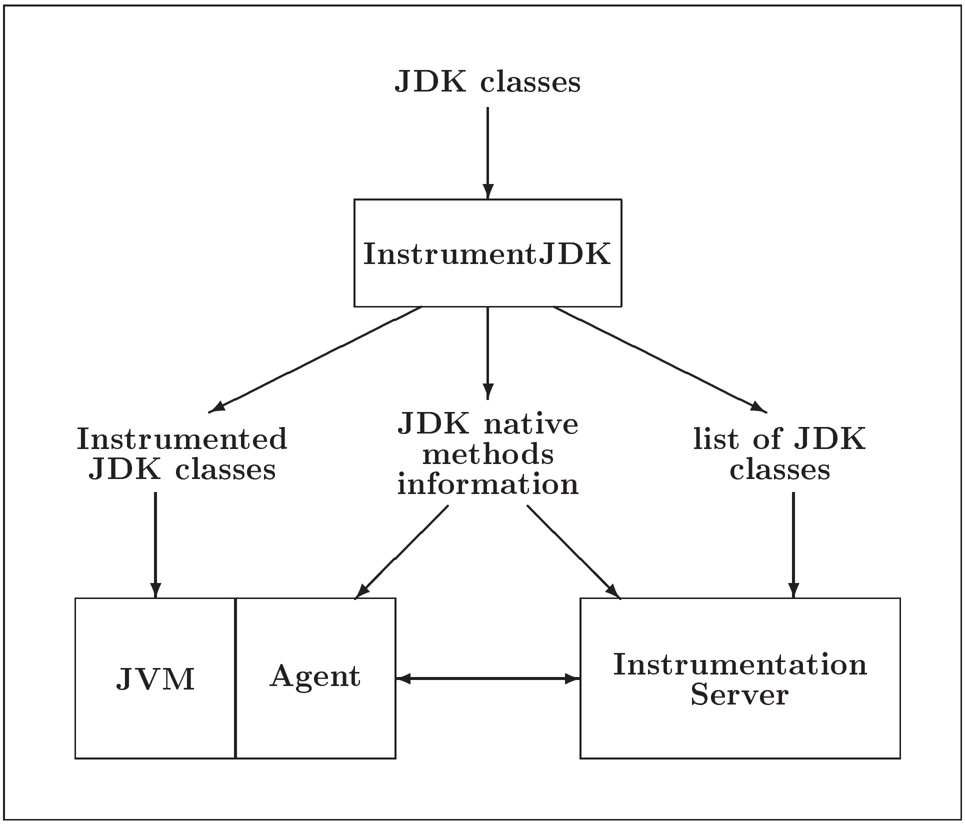

Figure 18: Instrumentation data flow.

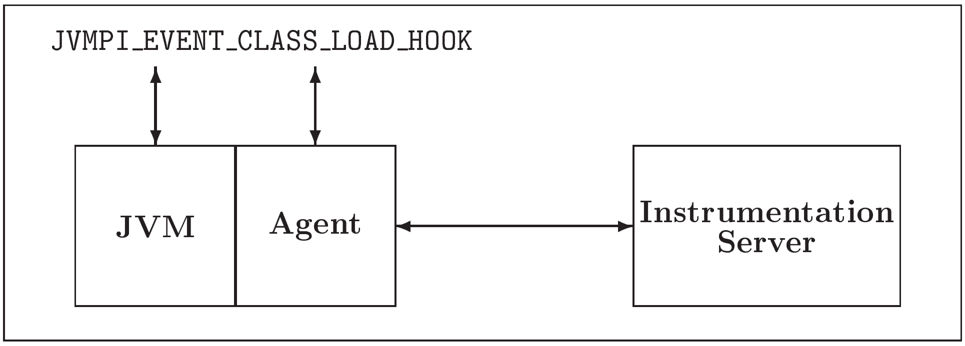

Figure 19: Dynamic instrumentation data flow.

through the `JVMPI_EVENT_CLASS_LOAD_HOOK` event, which is sent when the JVM obtains class file data, but before it constructs the in-memory representation for that class. The class is sent to the instrumentation server, which instruments it, and sends it back to the agent. Finally, the instrumented file is passed back to the JVM. This mechanism is illustrated in Figure 19. Information on JDK native methods and the list of JDK classes are passed to the instrumentation server, to allow the server to instrument application classes and to avoid instrumenting JDK classes.

The dynamic instrumentation causes overhead during the execution of the application, but for several reasons is still preferable over static instrumentation. First, some classes can be loaded from network; thus, they are not available before the application executes. There could also be a class loader in the application, which dynamically creates classes. Finally, as opposed to static instrumentation, the dynamic instrumentation does not require interaction with the user in order to find application classes.

6 The Profiling Agent

The profiling agent runs in the scope of the JVM and uses JVMPI events to collect profiling information. Table 8 presents the JVMPI events, which are used by the profiling agent.

Events are sent in the same thread where they are generated. For example, a class loading event is sent in the same thread in which the class is loaded. Multiple events may arrive concurrently in different threads. The agent program must therefore provide the synchronization necessary to prevent data corruption caused by multiple threads updating the same data structure at the same time.

Event	Description
JVMPI_EVENT_OBJECT_ALLOC	Sent when an object is allocated.
JVMPI_EVENT_OBJECT_FREE	Sent when an object is freed.
JVMPI_EVENT_OBJECT_MOVE	Sent when an object is moved in the heap.
JVMPI_EVENT_GC_START	Sent when GC is about to start.
JVMPI_EVENT_GC_FINISH	Sent when GC finishes.
JVMPI_EVENT_CLASS_LOAD_HOOK	Sent when the JVM obtains class file data, but before it constructs the in-memory representation for that class. The profiler agent can instrument the existing class file data sent by the JVM to include profiling hooks.
JVMPI_EVENT_CLASS_LOAD	Sent when a class is loaded in the JVM.
JVMPI_EVENT_CLASS_UNLOAD	Sent when a class is unloaded.
JVMPI_EVENT_METHOD_ENTRY2	Sent when a method is entered.
JVMPI_EVENT_JVM_INIT_DONE	Sent by the VM when its initialization is done.
JVMPI_EVENT_JVM_SHUT_DOWN	Sent by the VM when it is shutting down.

Table 8: The JVMPI events which are used by the profiling agent.

A class load event contains the class name, source file name, method information and instance field information. JVMPI defines a unique ID for every class, method and field. When the load of `java.lang.Object` is reported, the agent stores the IDs of the `__hup__GetOperation` and `__hup__PutOperation` methods. These IDs are used to identify the *put* and *get* types of object usage. In addition, the agent stores the IDs of constructors for all classes in order to exclude calls to these methods from object uses.

Object information is stored in a global in-memory database, where each database record represents one object. The database is currently implemented as an AVL tree (e.g., [Wei97]). In the future, we plan to change database implementation to hash table (e.g., [CLR90]). When a thread queries or updates the database, it locks the database to avoid data corruption. We are aware that this locking of the database can change application behavior. We discuss this issue in Section 8.1.

Object creation is reported by the `JVMPI_EVENT_OBJECT_ALLOC` event. The information in this event contains the object size, the class ID and the object ID, where the size includes the object header and alignment [VL00]. At this point, the agent creates a record in the database and stores the reported object information and the allocation time in it. Records in the database are marked with unique IDs, which are defined by JVMPI.

All object uses are reported by `JVMPI_EVENT_METHOD_ENTRY2` events. The `JVMPI_EVENT_METHOD_ENTRY2` event contains an object ID and a method ID. The first object usage is counted, when the first method invocation is reported and the method is not the object constructor and not the `__hup__PutOperation`. In order to count the last object usage, the agent updates the object record each time the `JVMPI_EVENT_METHOD_ENTRY2` event is fired for a given object and the reported method is not `__hup__PutOperation`. Object first and last usage information includes the time of usage and may also contain its stack trace at the time of use. In order to retrieve the stack trace, the agent calls the `GetCallTrace` JVMPI function. The agent can optionally save the stack traces for the first and last *put* operations on object.

There is no event in JVMPI interface that tells when an object becomes unreachable. To overcome this problem, the agent uses the `JVMPI_EVENT_OBJECT_FREE` event, which is fired each time the GC frees an object. It is not guaranteed that GC runs exactly at the point the object becomes unreachable. In order to increase the precision of this measurement, the agent triggers GC every 100Kb of allocation. Thus, the deviation from the point an object becomes unreachable does not exceed 100Kb. For simplicity, the agent reports object allocation, first and last usage times as the number of GC's since the start of the run.

During the run, the profiling agent logs the profiled data to file on disk.

An object's data is written into the file when the object is freed by the GC or when the `JVMPI_EVENT_JVM_SHUT_DOWN` event is received.

7 Related Work

This thesis continues the work of Shaham, Kolodner and Sagiv [SKS01], which instrumented the Sun reference JVM to record dragged objects and introduced analysis techniques and program transformations to reduce drag for Java programs. The additional contributions of this thesis can be summarized as follows:

1. The HUP tool is JVM independent and largely platform independent.
2. The HUP tool supports many additional features such as measuring lagged objects and new sophisticated analyses.
3. The tool is publicly available.
4. We designed an additional heap-reducing program transformation, which reduces array length (see Section 3.1).
5. We applied the tool to application programs that use the heap extensively and show that it can significantly improve heap consumption and reduce the maximum heap size (See Section 3).
6. We studied several definitions of dragged and lagged objects and their effects on the usability of the tool.

Röjemo and Runciman performed similar measurements of lagged, dragged and void heap space for Haskell language [RR96]. However, due to the fact that Haskell is functional and lazy, some of the techniques they used to reduce the heap consumption can not be used in Java and vice versa.

There are several tools for profiling Java programs. A survey of these tool is presented in [BG01]. None of these tools provide facilities for tracking of lagged, dragged and void objects.

In this thesis we used the manual rewriting of source code in order to reduce heap consumption. Alternatively, compilation techniques may be used to automate this task. For example, liveness analysis can be integrated with GC, so that reference variables with no future use (i.e., dead references) are not regarded as part of the root set [ADM98].

8 Conclusions and Future Work

In this thesis, we presented a useful profiling tool for investigating heap memory behavior and saving space. Our experiments indicate that it is quite easy to use the HUP tool in order to reduce the space of a given application. We believe that further investigation of lagged, dragged and void objects with the HUP tool in additional applications could inspire the development of new automatic space saving techniques for compilers and GC algorithms.

8.1 Limitations

One of the most serious drawbacks of the HUP tool is its runtime overhead. A profiled application may run fifty times slower with HUP, than it runs without it. The code that is injected by the instrumentation and JVMPI slow down the execution 8 times on average. Improvement of JVMPI in the future JVM implementations could solve these problems by decreasing the runtime overhead of JVMPI and providing more profiling information, which will allow us to avoid instrumentation of class files.

Another reason for slow application execution with the HUP tool, is the inefficient design of the profiling agent. The profiling agent stores object information in the global database, which is locked at each access to its records. The search in the database and its locking mechanism slow down the execution close to 6 times. Moreover the database locking mechanism adds locks, which are not present in the original program code. These locks can change the behavior of the profiled application or even create deadlocks. Independent logging of profiling information in each of the JVM threads could solve some of these problems. For example, Reiss and Renieries propose to generate multiple data streams from the JVMPI calls, one per thread, and then merge these data streams into a single stream in an independent process [RR00]. This approach does not introduce synchronization points and moves the processing of profiling data out of the JVM process.

The specification of the JVMPI does not explicitly define its behavior in the presence of incremental or generational garbage collectors [Wil92]. For the current HUP implementation we assume that when we trigger GC, all unreachable objects are freed. This assumption holds for the JVM implementations, which are currently supported by the HUP tool, but it may not hold in future JVM implementations. In order to overcome this problem, we could implement the *deep* GC algorithm in HUP and run it on a heap image, which could be obtained by requesting the `JVMPI_EVENT_HEAP_DUMP` event from the JVMPI interface.

There are bugs in the JVMPI implementation in some JVM-s. For example, the Sun HotSpot JVM [Gri98] loads some classes before loading the profiling agent. This bug prevents the profiling agent from receiving `JVMPI_EVENT_CLASS_LOAD_HOOK` and `JVMPI_EVENT_CLASS_LOAD` events for classes that are loaded before its initialization. We believe that fixing of this bug will allow the HUP tool to run with the Sun HotSpot JVM.

8.2 Suggestions for future research

In the future, HUP-analysis could provide additional techniques for exploration of the profiling data. For example, HUP-analysis could filter some objects in order to simplify analysis. In this way, the objects of the class `java.lang.Class` can be discarded from the analysis, due to the fact that these objects have special rules for their collection. Another approach is observing the heap topology. For example, we could classify objects by the objects that reference them. Similarly, objects could be classified by the references they hold. These and other analysis techniques are the subject for future research and experimentation.

HUP currently causes a large slowdown to the profiled program, thus, limiting the program size that can be analyzed. This limitation could be partially overcome if we could develop techniques to predict memory consumption for a bigger input or larger run, based on the analysis of the same program with smaller inputs. For example, if dragged objects are allocated in some nested allocation site throughout the application execution and never freed, we could suspect that the number of these objects and, thereby, the drag space would increase on a longer run. This problem could be determined by observing the allocation and the drag time of the dragged objects of a given nested allocation site. Unfortunately, in some situations the nested allocation site might be not brought to the programmer attention due to the relatively small drag space it produces on a short run. In order to overcome this problem, more sophisticated techniques would need to be developed.

Another topic for the future research is the object usage definition. As we mentioned in Section 4.1, we use the *put/get* approach to define object usage. Additional alternatives could be proposed. For example, first object use could be defined by observing dependencies between object accesses in the *put/get* approach. Similar to the *truly-live* variable definition [GMW81], we can say that an object is not used if it is accessed by a *put* operation or it is accessed by a *get* operation and the value that is retrieved by the *get* is used further only in *put* operations. Figure 20 illustrates this approach. The assignment `object.field_2 = object.field_1` is not counted as an object usage, because the value of `field_1` is used only in *put* operations.

```
object.field_1 = 0;
object.field_2 = object.field_1;
```

Figure 20: Example for extended *put/get* approach.

References

[ADM98] 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.

[AP00] C. Austin and M. Pawlan. *Advanced programming for the Java^{TM} platform.* Addison-Wesely, 2000.

[BG01] J. Bartolomé and J. Guitart. A survey on Java profiling tools. Research report UPC-CEPBA-2001-10, 2001.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* M.I.T. Press, 1990.

[Dah99] M. Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[GMW81] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *GI 11. Jahrestagung*, volume 50, pages 1–10, 1981.

[Gri98] D. Griswold. The Java HotSpot virtual machine architecture. *Sun Microsystems whitepaper*, 1998.

[IBM01] IBM. IBM JDK 1.3, 2001. Available at http://www.ibm.com/java.

[LAS00] T. Lev-Ami and S. Sagiv. TVLA: A framework for kleene based static analysis. In *SAS'00, Static Analysis Symposium*, pages 280–301. Springer, 2000. Available at "http://www.math.tau.ac.il/∼ rumster".

[LY96] T. Lindholm and F. Yellin. *The Java^{TM} virtual machine specification.* The Java Series. Addison-Wesely, 1996.

[RR96] N. Röjemo and C. Runciman. Lag, drag, void and use — heap profiling and space-efficient compilation revisited. *ACM SIGPLAN Notices*, 31(6):34–41, 1996.

[RR00] S. Reiss and M. Renieris. Generating java trace data. In *ACM 2000 conference on Java Grande*, pages 71–77, 2000.

[SKS00] R. Shaham, E. Kolodner, and M. Sagiv. On the effectiveness of GC in java. In *Int. Symp. on Memory Management*, pages 12–17. ACM, October 2000.

[SKS01] R. Shaham, E. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 104–113. ACM, June 2001.

[SPE98] SPECjvm98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.

[SUN01] SUN. Sun JDK 1.3, 2001. Available at http://java.sun.com/j2se.

[VL00] D. Viswanathan and S. Liang. Java virtual machine profiler interface. *IBM Systems Journal*, 2000.

[VRHS+99] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[Wei97] M. A. Weiss. *Data Structures and Algorithm Analysis in C.* Addison-Wesely, 1997.

[Wil92] P. R. Wilson. Uniprocessor garbage collection techniques. In *Memory Management, International Workshop IWMM*, September 1992.

A User's Manual

HUP is a heap-profiling tool that allows the exploration and reduction of heap space consumption in Java applications. The space saving is based on the fact that some of the allocated objects not immediately used (or not used at all) in the application code. Also, there are objects, which are no longer in use, but remain in memory. The HUP tool allows a programmer to locate and remove memory bottlenecks, which are caused by unused objects.

Currently, HUP has been tested on the Windows 2000 operating system with the IBM JDK 1.3 and the Sun JDK 1.3 (classic JVM implementations). It does not currently work with HotSpot due to bugs in HotSpot's implementation of JVMPI. HUP is distributed under the GNU General Public License.

A.1 Installation

The HUP tool is distributed in a zip file (`HUP-`*version*`.zip`). In order to install the HUP tool, unzip it into directory into which you want to install HUP.

In order to run the HUP tool you must install the BCEL package on your computer. The BCEL package can be downloaded from `http://bcel.sourceforge.net` (choose the latest version to download). We have tested with version `4.4.1`.

The HUP tool requires several environment variables to be set.

- Set the `JDK_HOME` to point at your JDK installation.
- Set the `BCEL_HOME` to point at your BCEL installation.
- Set the `HUP_HOME` to point at your HUP installation.
- Add to the `PATH` variable the `%HUP_HOME%\bin` directory.

Before any further step, run the `instrumentJDK` batch file from the `%HUP_HOME%\bin` directory. The `instrumentJDK` instruments the Java runtime class files from the `%JDK_HOME%\lib\rt.jar`. The instrumentation does not change the original JDK class files, but creates a copy of them, thus, the instrumentation does not affect the execution of Java applications with the original JDK. The `instrumentJDK` is a time-consuming process and may take several minutes, but you need to run it only once.

A.2 Running the profiler

Now you are ready to run the HUP profiler. In order to run an application under the HUP profiler, start the application with the `hup` instead of the `java` executable. For example, in order to profile the application `test`, type ``` ``%hup defaults test'' ``` instead of ``` ``%java test'' ```. If you need to pass parameters to the profiled application, write them after the name of the profiled application. For example, ``` ``%hup defaults test param1 param2 ...'' ```. If you would like to pass parameters to the `java` executable, you need to write them right before the name of the profiled application. For example, ``` ``%hup defaults -Xnoclassgc test'' ```.

The HUP tool uses `-classic`, `-Xbootclasspath` and `-Xrun` parameters for the `java` executable. Changing these parameters could be unsafe. There could be other runtime settings for a JVM implementation that could influence the running of the HUP tool.

HUP can be invoked with its either default configuration or its configuration can be changed by setting option. In order to run HUP with the default configuration, you must write the `defaults` keyword right after the `hup`, for example: ``` ``%hup defaults test'' ```. Alternatively, instead of the `defaults` keyword, configuration options can be specified. The options must be specified in the following format: `option1=value1,option2=value2,...` For example, ``` ``%hup od=c:\test_results,sd=10 test'' ```. Currently available options are:

- so=`1` - suppress HUP output during profiling.
- od=*path* - output directory for profiling results (default is `hup_results`).
- sd=*depth* - depth of stack trace dumps (default is 5, minimum is 0 and maximum is 10).

During profiling, HUP triggers garbage collection every 100Kb of allocation. In order to notify the user of progress, the HUP prints the `[GC...]` message at each garbage collection invocation. The `so` option allows you to suppress this notification.

The default output directory is created under the current directory. The `od` option allows you to specify another output directory instead of the default one. For example, ``` ``%hup od=c:\test_results test'' ```.

During profiling, HUP collects information about object allocation and usage. Part of this information is the stack trace of the thread at which object allocation or usage occurs. A bigger value of stack trace depth option yields more precise information, but may significantly slow down the execution. Under regular conditions we recommend using the default stack trace depth.

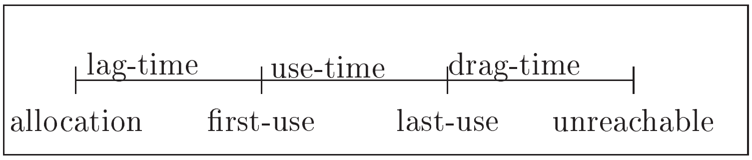

Figure 21: The lifetime of used objects.

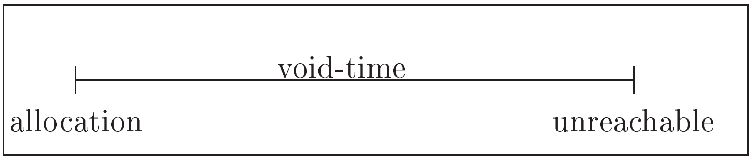

Figure 22: The lifetime of void objects.

A.3 Result analysis

First, we introduce some definitions, which are used in the following discussion. Generally, a Java program allocates objects and GC is responsible for collecting the objects, which are no longer in use and reclaiming their space. However, commonly used GC algorithms do not collect all potential garbage, rather just those objects that are no longer reachable from the *root set*. For example, there are objects that are reachable from the root set at a given point in the program and will not be used in the future. Some of these unused, but reachable objects could be reclaimed in order to save space. Moreover, on some occasions, we could delay the allocation of used objects, and thereby reduce heap consumption.

The lifecycle of an object is classified as shown in Figure 21. We refer to the time interval from the allocation time of an object until it is first used as *lag time* and to the object itself as a *lagged object*. The time interval from the last use of an object till it becomes unreachable is called *drag time* and object itself is said to be a *dragged object*.

In a special case, when the object has no uses at all, we refer to the interval between its allocation and the point it becomes unreachable as *void time* and the object itself as a *void object*, see Figure 22.

HUP measures the time in bytes allocated since the beginning of program execution. This provides a machine independent measure of time. Observing the size of reachable objects as function of time, we calculate the integral of the function. We refer to this space-time integral value as the *total space* of a given application. Similary, we refer to the values of the integral of lag, drag and void size functions as *total lag, drag and void space* respectively.

These definitions are particularly useful for understanding the impact of the lagged, dragged and void objects on an application heap consumption.

HUP-analysis is based on the classifications of lagged, dragged and void objects. Specifically, the analysis classifies the objects by class, allocation site and nested allocation site. The *nested allocation-site* of an object is the call chain of methods leading to the object allocation. In other words it is the thread stack trace at the point the object is allocated. Calling a method from different lines of code of the same method generate different nested allocation-sites. The *allocation site* of object is its nested allocation site of depth one, or simply the method, where the object is allocated. In contrast to the nested allocation site definition, the allocation site definition does not distinguish between the lines of the method. The analysis calculates lag, drag and void space for a given class, by summing the lag, drag and void space of all instances of this class, respectively. In the same way, the analysis calculates lag, drag and void space for allocation sites and nested allocation sites.

Another classification of objects that is provided by HUP-analysis is based on the differentiation of objects by their lifetime patterns. The lifetime pattern is defined by the first and the last object usage stack traces. For example, when you examine the lagged objects at a given nested allocation site, objects with different stack traces of first uses could be identified. This difference may point to different roles of the objects in a given program run, even though they are allocated at the same point. In the same way, you may explore the lifetime patterns of dragged objects.

One of the important issues is the definition of object usage. In the HUP tool, only read operations on objects are considered object usage. We refer to the read operations on an object as *get* operations and to the write operations on an object as *put* operations. Table 9 divides Java bytecodes into *put* and *get* operation groups. In this way, the first usage that is counted by the HUP tool can be preceded by *put* operations and the last usage that is counted by the HUP tool can be followed by *put* operations. In order to allow you to observe the object usage and decide for the right code transformation for space saving, the HUP provides stack traces for both first(last) put and get operations in lifetime patterns.

In order to begin the analysis process, invoke the `analysis` executable. The directory with the profiled results should be specified in the command line. Fot example: `"%analysis c:\test_results"`. The HUP-analysis loads and processes profiling results and then enters an interactive mode in which it receives and performs user commands. In following, we describe the currently available commands in HUP-analysis.

***get* operations**	***put* operations**
GETFIELD	PUTFIELD
AALOAD BALOAD CALOAD DALOAD FALOAD IALOAD LALOAD SALOAD ARRAYLENGTH	AASTORE BASTORE CASTORE DASTORE FASTORE IASTORE LASTORE SASTORE
INVOKEINTERFACE INVOKESPECIAL INVOKEVIRTUAL	
CHECKCAST INSTANCEOF	
MONITORENTER MONITOREXIT	
ATHROW	

Table 9: *put* and *get* operation groups.

- `help`
  The `help` command types the list of available commands.

- `write` *file*
  The `write` command tells the analysis to write the output of the next command into the specified *file*. For example, the command `"%write c:\help.txt"` followed by `"%help"` command will write list of the available commands into the `c:\help.txt` file.

- `stat [long]`
  The `stat` command prints the common statistics, such as the number of classes, the number of nested allocation sites and the number of lagged, dragged and void objects, which were determined in profiling results. It also prints the calculated *total space* of a given application and its *total lag, drag and void space.* The `long` parameter lists statistics for the garbage collector invocations. In particular, it prints the heap space size and the lag, drag and void space sizes for each invocation of the garbage collector.

- `obj` *id* `[long]`
  The `obj` command prints information for the object with identifier *id*: its class, its nested allocation site id and its lifetime pattern. The `long` parameter prints the stack trace at the point of object's allocation and stack traces for its first and last usage.

- `class` *id*|*name* `[long]`
  The `class` command prints class information; the class id, the corresponding class file name, the number of lagged, dragged and void objects of this class and the lag, drag and void spaces, which are generated by the objects of this class. The `long` parameter prints the id-s of the lagged, dragged and void objects of this class.

- `method` *id* `[long]`
  The `method` command prints method information: the method id, the method name, the corresponding class name, the number of lagged, dragged and void objects allocated by this method and the lag, drag and void spaces, which are generated by the objects allocated by this method. The `long` parameter prints the id-s of the lagged, dragged and void objects, which are allocated by this method.

- `nested` *id* `[long]`
  The `nested` command prints nested allocation site information: the

nested allocation site id, the number of lagged, dragged and void objects allocated at this site and the lag, drag and void spaces, which are generated by the objects allocated at this site. The `long` parameter prints the id-s of the lagged, dragged and void objects, which are allocated at this nested allocation site.

- `lag class [`*`id`*`] [@`*`num`*`]`
  `drag class [`*`id`*`] [@`*`num`*`]`
  `void class [`*`id`*`] [@`*`num`*`]`
  These commands print classes sorted by lag, drag and void respectively. If the class *id* is specified, the commands print list of the nested allocation sites in which objects of the specified class are allocated. The printed list is sorted by lag, drag or void. The number of nested allocated sites output can be limited by specifying the `@`*`num`* parameter.

- `lag method [`*`id`*`] [@`*`num`*`]`
  `drag method [`*`id`*`] [@`*`num`*`]`
  `void method [`*`id`*`] [@`*`num`*`]`
  These commands print methods sorted by lag, drag and void respectively. If the method *id* is specified, the commands print list of the nested allocation sites in which the specified method appears at the end of the call chain of methods leading to the object allocation. The printed list is sorted by lag, drag or void. The number of nested allocated sites output can be limited by specifying the `@`*`num`* parameter.

- `lag nested [`*`id`*`] [@`*`num`*`]`
  `drag nested [`*`id`*`] [@`*`num`*`]`
  `void nested [`*`id`*`] [@`*`num`*`]`
  These commands print nested allocation sites sorted by lag, drag and void respectively. If the nested allocation site *id* is specified, the commands print list of the lifetime patterns, which are found at the specified site. The printed list is sorted by lag, drag or void. Each lifetime pattern is printed with a representative object id. This id allows you to observe the lifetime pattern's stack traces by ``` ``obj ```*`id`*``` long'' ``` command. The number of lifetime patterns output can be limited by specifying the `@`*`num`* parameter.

- `exit`
  The `exit` command closes the HUP-analysis.

A.4 GNU Generic Public License

```
Version 2, June 1991

Copyright (c) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute

such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE

PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS