HUP: A Heap Usage Profiling Tool for Java Programs

–

User's Manual

Michael Pan

November 20, 2001

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.

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.

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).

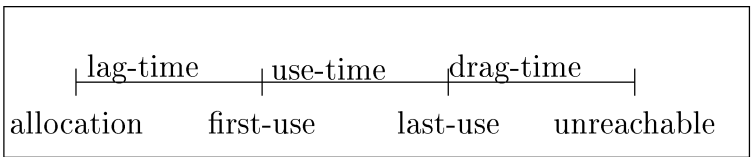

Figure 1: The lifetime of used objects.

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.

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 1. 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*

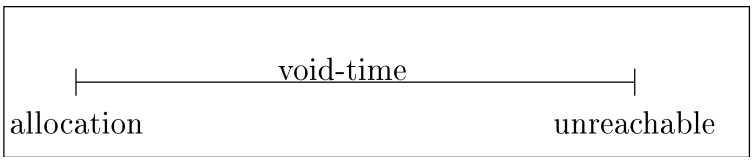

Figure 2: The lifetime of void objects.

time and the object itself as a *void object*, see Figure 2.

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

***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 1: *put* and *get* operation groups.

to the read operations on an object as *get* operations and to the write operations on an object as *put* operations. Table 1 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.

- `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.