

3ICT3: Computer Science, Profiling JAVA Programs

Dr. Andy Nisbet, ORI.LG.19, 3682, Andy.Nisbet@cs.tcd.ie

March 5, 2002

1 Objective

To gain an understanding of profiling in the JAVA environment using HPROF.

- Profiling concepts – sampling, versus timers.
- JAVA HPROF specifics.

For a more in depth discussion of profiling see the full text of:— http://www.usenix.org/publications/library/proceedings/coots99/full_papers/liang/liang_html/paper.html. which describes profiling as implemented in JDK1.2. This document contains sections of the paper mentioned above.

2 What is Profiling?

Profiling means the ability to monitor and trace events that occur during run time, the ability to track the cost of these events, as well as the ability to attribute the cost of the events to specific parts of the program. For example, a profiler may provide information about what portion of the program consumes the most amount of CPU time, or about what portion of the program allocates the most amount of memory.

- The profiler may measure the amount of CPU time consumed by a given method in a given class. In order to pinpoint the exact cause of inefficiency, the profiler may need to isolate the total CPU time of a method **A.f** called

from another method **B.g**, and ignore all other calls to **A.f**. Similarly, the profiler may only want to measure the cost of executing a method in a particular thread.

- The profiler may inform the programmer why there is excessive creation of object instances that belong to a given class. The programmer may want to know, for example, that many instances of class **D** are allocated in method **C.h**. More specifically, it is also useful to know that majority of these allocations occur when **B.g** calls **C.h**, and only when **A.f** calls **B.g**.
- The profiler may show why a certain object is not being garbage collected. The programmer may want to know, for example, that an instance of class **C** is not garbage collected because it is referred to by an instance of class **D**, which is then referred to by a local variable in an active stack frame of method **B.g**.
- The profiler may identify the monitors that are contended by multiple threads. It is useful to know, for example, that two threads, **T1** and **T2**, repeatedly contend to enter the monitor associated with an instance of class **C**.
- The profiler may inform the programmer what causes a given class to be loaded. Class loading not only takes time, but also consumes memory resources in the Java virtual machine. Knowing the exact reason that a class is loaded, the programmer can optimize the code to reduce memory usage.

3 The HPROF Agent

The JDK provides an interface known as the JVMPI – JAVA virtual machine profiler interface. The HPROF agent is a dynamically-linked library shipped with JDK 1.x. It interacts with the JVMPI and presents profiling information either to the user directly or through profiler front-ends.

We can invoke the HPROF agent by passing a special option to the Java virtual machine:

java -Xrunhprof ProgName

ProgName is the name of a Java application. Note that we pass the -Xrunhprof option to java, the optimized version of the Java virtual machine. We need not rely on a specially instrumented version of the virtual machine to support profiling. NOTE: for more information on the specific options available on your platform type **java -X**.

Depending on the type of profiling requested, HPROF instructs the virtual machine to send it the relevant profiling events. It gathers the event data into profiling information and outputs the result by default to a file. For example, the following command obtains the heap allocation profile for running a program:

```
java -Xrunhprof:heap=sites ProgName
```

3.1 HPROF Stack Traces

Each frame in the stack trace contains class name, method name, source file name, and the line number. The user can set the maximum number of frames collected by the HPROF agent. The default limit is 4. Stack traces can be used to reveal not only which methods performed heap allocation, but also which methods were ultimately responsible for making calls that resulted in memory allocation.

3.2 HPROF CPU Profiling

A CPU time profiler collects data about how much CPU time is spent in different parts of the program. Equipped with this information, programmers can find ways to reduce the total execution time.

There are two ways to obtain profiling information: either statistical sampling or code instrumentation. Statistical sampling is less disruptive to program execution, but cannot provide completely accurate information. Code instrumentation, on the other hand, may be more disruptive, but allows the profiler to record all the events it is interested in. Specifically in CPU time profiling, statistical sampling may reveal, for example, the relative percentage of time spent in frequently-called methods, whereas code instrumentation can report the exact number of time each method is invoked.

A less disruptive way to implement code instrumentation is to inject profiling code directly into the profiled program. This type of code instrumentation is easier on the Java platform than on traditional CPUs, because there is a standard class file format. The JVMPI allows the profiler agent to instrument every class file before it is loaded by the virtual machine. The profiler agent may, for example, insert custom byte code sequence that records method invocations, control flow among the basic blocks, or other operations (such as object creation or monitor operations) performed inside the method body. When the profiler agent changes the content of a class file, it must ensure that the resulting class file is still valid according to the Java virtual machine specification.

Thread-Aware CPU Time Sampling The Java virtual machine is a multi-threaded execution environment. One difficulty in building CPU time profilers for such systems is how to properly attribute CPU time to each thread, so that the time spent in a method is accounted only when the method actually runs on the CPU, not when it is unscheduled and waiting to run. Thread aware support for sampling can be turned on using the incantation **java -Xrunhprof:thread=y Myprog**.

```
java -X
  -Xmixed              mixed mode execution (default)
  -Xint                interpreted mode execution only
  -Xbootclasspath:<directories and zip/jar files separated by ;>
                        set search path for bootstrap classes and resources
  -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                        append to end of bootstrap class path
  -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                        prepend in front of bootstrap class path
  -Xnoclassgc          disable class garbage collection
  -Xincgc              enable incremental garbage collection
  -Xms<size>           set initial Java heap size
  -Xmx<size>           set maximum Java heap size
  -Xss<size>           set Java thread stack size
  -Xprof               output cpu profiling data
  -Xrunhprof[:help] |[:<option>=<value>, ...]
                        perform JVMPI heap, cpu, or monitor profiling
  -Xdebug              enable remote debugging
```

The -X options are non-standard and subject to change without notice.

```
java -Xrunhprof:help
Hprof usage: -Xrunhprof[:help] |[:<option>=<value>, ...]
```

```
Option Name and Value Description Default
-----
heap=dump|sites|all heap profiling all
cpu=samples|times|old CPU usage off
monitor=y|n monitor contention n
format=a|b ascii or binary output a
file=<file>write data to file java.hprof(.txt for ascii)
net=<host>:<port>send data over a socket write to file
depth=<size>stack trace depth 4
cutoff=<value>output cutoff point 0.0001
lineno=y|n line number in traces? y
```

```
thread=y|n thread in traces? n  
doe=y|n dump on exit? y
```

Example: `java -Xrunhprof:cpu=samples,file=log.txt,depth=3 FooClass`

4 Experiments

1. Download and compile the file **Profile.java**, examine the program, and understand its operation.
2. Run the program without profiling and note its duration (time).
3. Experiment with profiling using HPROF, especially with respect to CPU time based on **times** and **samples**. What effect does profiling have on the execution time?
4. What is the effect of turning **thread** support on?
5. Experiment with HPROF using the **heap**.
6. Examine the source code again, and repeat your experiments after performing the suggested comment alterations (see java source file).