Volume 6, Issue 3, March 2016 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Algorithm Analysis-A Technical Report

[1]Ritu*, [2]Yatika Hasija
[1] Assistant Professor, Computer Science & Engineering, Chandigarh Group of College, Landran, Mohali, India
[2] Assistant Professor, Information Technology, ITFT, Chandigarh, India

Abstract—traditionally, the emphasis among computer scientists has been on the more rigorous and theoretical modes of worst-case and average-case analysis of algorithms. However, theoretical analysis cannot tell the full story about real-world performance of algorithms. This has resulted in a growing interest in experimental analysis. This paper presents the development of a solution that aide in the experimental analysis of Java-based implementations.

Keywords—Best; Worst; Bound;

I. INTRODUCTION

We identify two roles with regard to performing experimental analysis of algorithms: the Experimenter and the Analyst. The Experimenter performs the experiments and the Analyst analyzes the result of the experiments. The role of Experimenter can be automated and is done soin this paper. The software developed, the Timing API, allows the Analyst to create the test data and input this to the Timing API. The API carries out the experiments and records the results for later analysis by the Analyst. This removes the tedium of developing software to test implementations, running the tests and recording the results of the test essentially carrying out the tasks of the Experimenter. The paper is structured as follows. Section 2 gives a brief overview of theoretical Algorithm Analysis. Following that, Section 3 mentions Experimental Algorithm Analysis and some of the issues involved in performing such analysis. Sections 4 and 5 describe the design and implementation of the Timing API developed to automate the role of the Experiment. The Java Grande Forum Benchmark is covered in Section 6 along with details of improvements made by the author to the benchmark. Section7 provides an example illustrating the use of the Timing API on another Java-based API. Conclusions and future work are discussed in Section 8.

II. THEORETICAL ALGORITHM ANALYSIS

The complexity of an algorithm is the cost of using the algorithm. Space
Complexity is the amount of memory needed. Time complexity is the amount of computation time needed to run the algorithm. Complexity is not an absolute measure, but rather a bounding function characterizing the behavior of the algorithm as the size of the data set increases. This leads to the Big-Oh Notation which is briefly defined here. For a more comprehensive introduction, see [Wilf, 1994].
Upper Bound, Lower Bound, Tight Bound.

III. EXPERIMENTAL ALGORITHMS ANALYSIS

There are three different approaches to analyzing algorithms.

1. Worst-Case Analysis
2. Average-Case Analysis
3. Experimental Analysis

Johnson's paper discusses how best to perform experimental analysis of algorithms. The author discusses problems of performing this type of analysis and how to overcome them. These include
Pitfalls: Defined by Johnson as"temptations and practices that can lead experimenters into substantial wastes of time".

a. Lost code/data
b. Loss of the original code and/or data means
c. the loss of the best approach to comparing it and future algorithms.

Pet Peeves: "Favorite annoyances" of Johnson. These are common practices that are misguided.

a. The millisecond testbed Regardless of the resolution of timings they cannot be more precise than the resolution of the underlying operating system.
b. The one-run study A one-run study is where the tests are run once only as opposed to the preferred method of running tests many times to reduce errors.

IV. THE TIMING API DESIGN

The design of the Timing API is heavily influenced by existing implemen- tations for performing unit tests.

4.1 Use Case Model

Use cases document the behavior of the system from the user's point of view [Stevens and Pooley, 2000]. An individual use case, shown as a named oval, represents a kind of task which has to be done with support from the system under development. An actor, shown as a stick person, represents a user of the system. A user is anything external to the system Which interacts with the system? The use case model (Figure 1) reveals that there are two actors in performing experimental analysis of algorithms. The first actor, Experimenter, is the sole user of the Timing API. The tasks of the other actor, Analyst, are outside the scope of the API.

4.2 Class Model

Class models are used to document the static structure of the system [Stevens and Pooley, 2000]. Figure 2 contains the class model for the Timing API. The Time Runner class is the driver class that loads in the con figuration files and passes them onto the Time Configuration class. This class in turn creates a Time Suite Configuration instance which contains information such as the threshold and the stopwatch - both of which will be discussed in greater detail further on - to be used by the Timing API. Time Suite instances contain the unit times. A unit time is defined below as being the execution time of a single unit of code. Each unit time produces a Time Report containing the results of the experiment on that unit time. The Time Report instance will be recorded using result reporting.

V. THE TIMING API IMPLEMENTATION

The implementation consists of three main components.

a) Timing Tests
b) Timing Mechanism
c) Result Reporting

Each of these components is discussed here in turn.

5.1 Timing Tests

Unit tests are so named because they test a single unit of code typically a method or function. One or more unit tests are contained within a suite. Furthermore, suites can be contained within other suites as sub-suites. Similarly, we define a unit time to be the execution time of a single unit of code (Figure 3).Atria's Suite Runner [Venners et al., 2003] is an architecture for writing unit tests that offers advanced facilities for reporting results of the unit tests. It was hoped that an extension to Suite Runner would be sufficient for the purposes of unit timing. Preliminary work demonstrated otherwise as a significant amount of the Suite Runner classes are package protected making extensions to it difficult. As a result, the Timing API code base was developed from scratch. The Experimenter extends the unit time class.TimeSuit to create a timing test as follows:

```
Import nuim.cs.crypto.tamull.TimeSuite;
Public class MyTimeSuite extends TimeSuite {
Public void timeMyMethod () {
MyMethod () ;}}
```

Note that the code unit - typically a method itself - being timed must be placed within a method of a particular signature with return type void. The signature of a method is a combination of the method's name and its Parameter types. In this case, the signature is a method beginning with the letters time and consisting of no parameters. The Seek API (Figure 4) contains functionality for extracting all methods of a particular signature and return type. This functionality is contained within the Method Seeker class. The signature of the method and the return type has to be specified by the developer using the API. Thus, the Seek API is used by the Timing API to extract the types of methods described in the preceding paragraph. See appendix B.2.2 for an example of the Seek API in action. The Class Seeker class is not directly used by the Timing API but it can still play a role. For example, it may be used to find all the classes that are subclasses of the Time Suite class. These classes can then be passed into the Timing API and the relevant methods extracted as explained before. By saving the code and the input and output files used, the Experimenter addresses Johnson's Pet Peeve 5 of the lost testbed and Pitfall 1 of lost code/data

This allows for reproducibility of the results obtained and for comparisons to be made in the future against the implemented algorithms and possible improvements of the same algorithms."Just as scientists are required to preserve their lab notebooks, it is still good policy to keep the original data if you are going to publish results and conclusions based on it" [Johnson, 2002].

It is worth noting that regardless of the resolution of the stopwatches (milliseconds for the System Stopwatch and nanoseconds for the Thread Stopwatch) they cannot be more precise than the underlying operating system. On Windows NT/2000, the granularity is 10 milliseconds and on Windows 98 it is 50 milliseconds. For more information see [java.sun.com, 2001]. A threshold parameter is used to ensure that all tests exceed the granularity of the operating system by a significant amount answering Pet Peeve 1 of the millisecond testbed and Pet Peeve 6 of false precision. The test is run repeatedly until the threshold is reached. The total running time and the number of times the algorithm was executed is reported, not just the average running time. This requirement is in response to Pet Peeve 7 of failure to report overall running time.

5.1.1 System Stopwatch

This stopwatch uses the textbook approach to simple performance analysis

In Java by calling the System.currentTimeMillis () method before and after the code to be measured. The difference between the two times gives the Elapsed time. This is comparable to using a stopwatch when testing GUI Activity and works fine if elapsed time is really what you want. The downside is that this approach may include much more than your code's Execution time. Time used by other processes on the system or time spent
Waiting for I/O can result in inaccurately high timing numbers.

5.1.2 Thread Stopwatch

This stopwatch is useful for obtaining elapsed timings for a thread. The Java Virtual Machine Profiler Interface (JVMPI) provides the correct information on CPU time spent in the current thread accessed from within a Java program. This avoids the downside of using the System Stopwatch mentioned in the previous paragraph. This involves the use of the Java Native Interface (JNI) API. The basis for this particular stopwatch can be found at [Gørtz, 2003.

5.2 Result Reporting

The Reporter API uses the Observer design pattern (Figure 6). Note, in the diagram the word Observable may be substituted for the word Subject.The intent of the design pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents ar notified and updated automatically. More details of the design pattern can be found in [Cooper, 1998].In the case of the Reporter API, the Observer is the DispatchReporter and the Subjects are the Reporters. A message is sent to the Dispatch Reporter which then relays that message to the individual Reporters allowing for the same message to be reported at the same same time in many different media (Figure 7). Examples of such reporters are

- Console Display
- GUI Display
- Flat Text File
- XML File
- Relational Database

The Timing API uses the Reporter API to record the timing results of the tests and of the benchmarks. Good reporting avoids Pitfall 1 of lost code/data and repeated runs of the tests address Pet Peeve 2 of the one-run study and Pet Peeve 3 of using the best result found as an evaluation criteria. By using the Observer design pattern, the results can be simultaneously displayed on screen and in one or more files of differing formats. The displaying of results on screen provides feedback to the experimenter as to the progress of the tests, see Figure 8. The reporter API allows the experimenter to select the desired file format such as flat file, XML, CSV (Comma-Seperated Value), etc. without specific code being included in the Timing API for each format.

5.3 Timing API Extension

There are two factors when timing a unit of code - the algorithm and the size of the input to the algorithm. In cases where a large number of algorithms and/or input data are being timed, it is prohibitive to write code for each timing to be performed. To deal with this, the Timing API has been extended to make use of the Reflection API [Green, 2003] to allow methods and input data to be contained within an XML file. Each timing is performed and recorded and the corresponding entry in the XML file is processed. See Appendix B.2 for more details.

VI. BENCHMARKS

There is many Java Virtual Machine benchmarks such as the SPEC JVM98 and Java Grande Forum (JGF) Sequential Benchmark. The JGF benchmark was selected due to the availability of the source code which allows users to understand exactly what the benchmark is testing. Also, the JGF benchmark is available free of charge. The purpose of the JGF Benchmarks are to test aspects of Java execution environments [Bull et al., 2000a]. While there are several different JGF Benchmarks, the Sequential Benchmark (version 2.0) is sufficient for timing tests related to sequential, as opposed to multi-threaded or distributed, software. Version 2.0 of the Sequential Benchmark - as developed by the EPCC of the University of Edinburgh on behalf of the JGF consists of 3 sections.

1. **Low Level Operations**. This section is designed to test low-level operations such as the performance of arithmetic operations, of creating objects and so on.

2 Kernels. Created for testing short codes likely to be found in Java Grande applications such as Fast Fourier Transform.

3 Applications. This section is intended to be representative of Grande applications by containing actual Grande applications. A Grande application is one which uses large amounts of processing, I/O band-width, or memory. In this instance, the JGF Sequential Benchmark is used to calibrate the machine before running timing tests, see Pet Peeve 4 (the uncalibrated machine) and Suggestion 2 (use benchmark algorithms to calibrate machine speeds)."Future researchers can the calibrate their own machines in the same way and, based on the benchmark data reported in the original paper, attempt to normalize the old results to their current machines" [Johnson,2002].However, some alterations were necessary to the JGF Sequential Benchmark in order to satisfy the Johnson's requirements for experimental analysis. The shortcomings were

1. Lack of configuration. The experimenter had to run each suite or section separately as a driver class was required for each. To run a custom made selection of the benchmark tests required the development and compilation of new code - a cumbersome and overly complicated approach.

2. Insufficient reporting of results. The results of the benchmark tests were only reported to the console only. It was necessary to cut-n-paste the results to a text file before any analysis could be performed.This may result in the loss of data.

3. Result interpretation. Only the performance value of the benchmark tests are reported on completion of the test. The performance value is a derived value, based on the time and opcount values. This approach is not in keeping with scientific standards of observation and measurement.

4. Interrupt unfriendly. If the benchmark were interrupted due to say power failure the tests would have to be restarted from the beginning. Furthermore, the machine that the benchmark is running on is off-limits until the completion of the benchmark as other running processes may affect the result. This is a problem considering that the entire benchmark can take several hours to complete.

6.1 About JGF Numbers

For each section of the benchmark suite we calculate an average performance relative to a reference system, the result of which is referred to as the JGF number. This system, as selected by the authors of Java Grande, is an Sun Ultra Enterprise 3000, 250 MHz, running Solaris 2.6 and using Sun JDK 1.2.1 02 (Production version). The JGF numbers are calculated using the geometric mean as follows where pc is the performance value achieved on the system being bench- marked, where ps

is the performance value for the benchmark of the reference system and where I is the ith benchmark and b is the number of benchmarks in the section. ”Averaged performance metrics can be produced for groups of benchmarks, using the geometric mean of the relative performances on each benchmark”[Bull et al., 2000b]. For the machine used by the author, the JGF number was found to be 2.59824 for Section 1, 4.97861 for Section 2 and 6.57787 for Section 3. To clarify, low level operations (Section 2) on the author’s machine are over two and a half times greater than that of the reference system. See appendix A.1 for more details of the machine used by the author and appendix A.2 for the details of the JGF numbers quoted.

6.2 Configuration

The benchmark XML configuration file allows for custom made selection of benchmark tests to be run without the development of new driver code.The configuration file has the following format

```
<benchmarks>
<benchmarks>
<classname>uk.ac.ed.epcc.jgf.section3.euler.JGFEulerBench</classname>
<size>A</size>
</benchmark>
</benchmark>
```

where each classname is the name of a class that is a subclass of one of the following

1. uk.ac.ed.epcc.jgf.jgfutil.JGFSection1
2. uk.ac.ed.epcc.jgf.jgfutil.JGFSection2
3. uk.ac.ed.epcc.jgf.jgfutil.JGFSection3

Subclasses of JGF Section1 are Low Level Operations, subclasses of JGFSection2 are Kernels and subclasses of JGFSection3 are Applications. A range of data sizes for each benchmark in Sections 2 and 3 is provided to avoid dependence on particular data sizes.

6.3 Result Reporting

Using the reporting facilities of the Timing API allows for results to be both displayed on the console and stored in an XML file. The XML file also contains details such as the operating system name and version and details of the JVM that the benchmark was executed on. The new design also ensures that only one result for each benchmark (i.e., the most recent result) is stored into the specified XML file, irrespective of the number of times the same benchmark is executed. It is possible for other reporting methods to be appended at a later date. The XML report file has the following format

```
<?xml version="1.0" encoding="UTF-8"?>
<benchmarks
name="Java Grande"
type="Sequential"
version="2.0"
os_name="Windows 2000"
os_arch="x86"
os_version="5.0"
java_vm_specification_version="1.0"
java_vm_specification_vendor="Sun Microsystems Inc."
java_vm_specification.name="Java Virtual Machine Specification"
java_vm_version="1.4.0_01-b03"
java_vm_vendor="Sun Microsystems Inc."
java_vm_name="Java HotSpot(TM) Client VM">
```


<benchmark section="Section1" suite="Arith" name="Add:Int" size="-1">
<time>10531</time>
<opname>adds</opname>
<opcount>1.048576E10</opcount>
<calls>1</calls>
</benchmark>
</benchmarks>

6.4 Result Interpretation

The time required by the benchmark is now stored in milliseconds. This means that the output has not been altered in any way in keeping with scientific standards of observation and measurement. The performance value of a benchmark is derived information, thus it is not reported. Temporal performance is defined in units of operations per second, where the operation is chosen to be the most appropriate for each individual benchmark

Performance = Operation/Time

This calculation belongs in the analysis of the results and not included in the reported result. As the benchmark results are now stored in XML it is easier to interpret and present the results. Existing code to convert a text file containing results into an html file is unnecessarily complicated. With the modifications detailed above, the process is now simpler. The introduction of templates (such as the freemarker template JGFBenchmark.html found in the template directory) allows for changes to be made to the presentation

of results without modifications to code. This also allows for possibility of the results being displayed in a format other than HTML with the creation of an appropriate freemarker template. Futhermore, the calculation of the JGF number has been separated from the presentation logic into a component of its own. Consequently, the interpretation and presentation of the results are now two separate components (JGFBenchmarkNumber.java and JGFBenchmarkPrinter.java respectively).

6.5 Restart Functionality

An XML-based status file is used to record the current progress of the benchmark run. This allows the current run to be interrupted and restarted without re-running benchmarks that have already been completed success- fully.

VII. AN EXAMPLE

7.1 The Polynomial Arithmetic API

Polynomials and polynomial arithmetic play an important role in many areas of mathematics. The Polynomial API was developed by the Crypto Group,Computer Science Department, National University of Ireland NUI) Maynooth for the purpose of performing arithmetic operations on unbounded univariate and bivariate polynomials [Burnett et al., 2003] (Figure 9).

A univariate (one-variable) polynomial of degree d has the following form where each term of the form cx is referred to as a monomial. Each monomial in a univariate polynomial has a coefficient c a variable x and an exponent e. The largest exponent of all the monomials in a univariate polynomial is referred to as the degree of the polynomial d.A bivariate (two-variables) polynomial has monomials of the form cxwhere c is the coefficient, two variables x and y. Each variable has its own exponent value

E and ey. The total degree of a bivariate monomial is the sum of ex and ey and the total degree d of a bivariate polynomial is the largest total degree value for the monomials in the polynomial.

7.1.1 Algorithm Analysis

Univariate Polynomial

The following are the results for the univariate polynomial class. In the results for the univariate polynomial class BigPolynomial, the variables M represents the number of monomials in the polynomial, d the degree of the polynomial and c the size of the largest coefficient in the polynomial.

1. Multiplication: There are two strategies available for univariate polynomial multiplication.

School Book. The method taught in school books, hence the name, where each term in one polynomial is multiplied with each term in the other in turn. This algorithm has a complexity of

O (m2Minteger (c2))

Where Minteger is the cost of multiply two large integers.

2. Binary Segmentation. This method ”amounts to placing polynomial coefficients strategically within certain large integers, and doing all the arithmetic with one high-precision integer multiply” [Crandall and Pomerance, 2001]. According to Crandall and Pomerance, this has a complexity

O (Minteger (dln (dc2)))

Where Minteger is the cost of multiply two large integers. Figure 10 shows that the Binary Segmentation algorithm offers significant performance advantage compared to the School Book algorithm. This is in keeping with the theoretical complexity analysis results above.

VIII. CONCLUSIONS

This paper detailed the development of the Timing API to automate the role of the Experimenter in performing Experimental Algorithm Analysis. It has created solutions to points raised in Johnson’s paper regarding Experimental

Algorithm Analysis and incorporated those solutions in the Timing API. Consequently, this API evaluates the real-time performance of different algorithms. The benchmarking carried out in this paper will allow others to normalize the results obtained to their own machines. Therefore, our API will not be adversely affected by different machine speeds or specifications. Future work will involve the use of the Timing API as a decision tool in selecting the most efficient component algorithms for use in elliptic curve cryptosystems. The example given measures the performance of one such component. The selection will be based on the type of curve selected and the underlying machine as demonstrated by our Polynomial Arithmetic.

REFERENCES

[1] Ant. Jakarta ant project.http://ant.jakarta.apache.org, 2003
[2] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java.Concurrency: Practice and Experience, pages 375–388, 2000a
[3] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. Benchmarking java grande applications. pages 63–73, 2000b
[4] Andrew Burnett, Adam Duffy, , Tom Dowling, and Claire Whelan.
[5] A java api for polynomial arithmetic. Principles and Practice of Programming in Java, 2003.
[6] James W. Cooper.The Design Patterns Java Companion Addison- Wesley, 1st edition, 1998.
[7] Richard Crandall and Carl Pomerance.Prime Numbers - A Computational Perspective. Springer-Verlag, 1st edition, 2001
[8] Adam Duffy. Xscribe api.http://www.ijug.org/member/articles/xscribe/2002.
[9] Benjamin Geer and Mike Bayer.Freemarker api. http://freemarker.sourceforge.net/, 2001
[10] Jesper Gørtz. Use the jvm profiler interface for accurate timing.
[11] http://www.javaworld.com/javatips/jw-javatif94.html, 2003.
[12] Dale Green.The java tutorial : The reflection api.
[13] http://java.sun.com/docs/books/tutorial/reflect/, 2003.

