JAVA BYTECODE INSTRUCTION USAGE COUNTING WITH ALGATOR

Tomaž DOBRAVEC
University of Ljubljana, Faculty of Computer and Information Science, Večna pot 113, 1000 Ljubljana, Slovenija
E-mail address: tomaz.dobravec@fri.uni-lj.si

ABSTRACT

Development of algorithms for solving various kinds of computer related problems consists of several consecutive and possibly repetitive phases. The final and very important step in this process is to implement the developed algorithm in a selected programming language, to test its behavior on some real-world test cases and to compare the results with the results of other algorithms. This evaluation can be done by comparing different execution indicators among which the time consumption is usually considered to be the most relevant. On the other hand, timing the algorithms in practice is very difficult since it is hard to ensure a fair and reproducible environment in which algorithm's implementations can be compared. To overcome this barrier, we introduce a system called ALGATOR *that was developed to facilitate the algorithm evaluation process. Besides the time complexity and the project-specific indicators,* ALGATOR *also measures the counters of Java code and Java bytecode usage. The measurement of the former is implemented by using special tags that are to be inserted in the appropriate lines of Java code while the measurement of the latest is enabled by using an adapted Java virtual machine, which counts the Java bytecode usage and reports the statistics. By using this counters new timing-independent criteria for algorithm assessment can be derived. In this paper we present some basic concepts of the* ALGATOR *system and give some examples of how to use the system in practice. We show the distribution of the usage of Java bytecode instruction for the sorting problem and the usage of the Java bytecode indicators in the time-complexity prediction for the matrix multiplication algorithm. The examples presented in this article show how the classic time measurement methods can be replaced by measuring some other more reliable indicators, and how this measurements can help to asses the quality of our algorithms.*

Keywords: *Algorithm evaluation, Java bytecode counting, time-complexity prediction*

1. INTRODUCTION

The results obtained by the last two phases of the algorithm design process (i.e. the proof of the correctness and the estimation of the complexity) are based on the assumptions of the selected computational model, which (more or less successfully) imitates the real environment in which the algorithm will be implemented. From a theoretical point of view, these results are interesting and completely legitimate, since they allow (theoretically) to compare and classify algorithms. When applying these results in practice, however, problems may arise, as it often shows that the real environment differs from the theoretical assumptions. Thus, for example, theoretical models usually do not include assumptions about the concrete implementation of memory management (and more specifically, the influence of the cache), which in practice greatly affect the speed of implementation.
To choose the best algorithm for solving a given problem, theoretical results may help in the first round of selection, where the algorithms with the best asymptotical boundaries are selected. A real distinction between the selected algorithms with the same theoretical boundaries can only be made by comparing their behaviour in the real environment [3, 4]. Timing the algorithms in practice is very difficult since it is hard to ensure a fair and reproducible environment in which algorithms can be compared. The results of the measurements are influenced by various factors, some of them are more or less random. In order to better assess the practical time complexity, we, therefore, need additional tools to measure independent indicators of the implementation of algorithms.

In this paper we present a novel approach for measuring and predicting the complexity of the algorithms' implementations by counting the Java bytecode instructions [2]. In Section II. we present a tool called ALGATOR [1], which was designed to facilitate the algorithm comparison process by measuring different indicators. We also present the three types of measurements which are supported by the ALGATOR. In Section III. we focus on a simple problem of matrix multiplication and present different approaches to produce useful performance predictions based on the java bytecode instructions usages. We conclude this paper with the final remarks in Section IV.

2. THE ALGATOR

The ALGATOR is a computer application that was developed to support and to facilitate the algorithm design and evaluation process. The main entity within the system is the so called 'project' which it defined by a set of definitions for the problem, the algorithms and the test sets. The system was designed to be as general as possible and therefore applicable in a wide range of problem domains. All the entities (projects, algorithms, test cases) are primarily defined on an abstract level and the system is trained to execute abstract algorithms on abstract test cases. After selecting a real problem, user concretizes the abstract parts of the project and makes the project 'alive' and prepared to be used to execute real algorithms on real data.
The abstraction of the project is integrated in several parts of the ALGATOR system. The algorithm is defined as a block of code (e.g. a class in Java) with predefined hook used to pass the parameters and start the execution (e.g. a method signature). The test case and the results of the execution are defined as an arbitrary data structures implemented in a selected programming language. A test set, which represents a minimal execution unit, is composed of

several test cases and it is iterated through during the algorithm execution process. To collect and present the results of the algorithm execution, the ALGATOR uses the so called result sets (i.e. the sets of parameters and indications of the execution) and the presenters (the definition files in which the type and the range of the presentation is provided). All these abstractions make the system flexible and usable in many fields of computer science.

2.1. The ALGATOR project

To define an ALGATOR project, user must provide both, the configuration files and the source code in a selected programming language (Java, C or C++). The configuration files define administrative data (the name and the author of the project, the number of supported algorithms, the time limit for algorithm execution, ...) while the source code provide the logic for executing the algorithms and for evaluating the results of the execution. The configuration files use the `json` format and have predefined names and positions in the folder hierarchy of the project. For example, the configuration file for the project `P` is named `P.atp` and it is placed in the subfolder `proj` of the project folder `PROJ-P`. Besides the basic information about the project the configuration of the project also include the information about algorithms (the name and the author of each algorithm, programming language of the algorithm, ...), about test sets (the number of test cases of each test set, sizes of the test cases, time limits for execution of a test case, ...) and about the results (the number and the type of the indicators of execution).

The source code of the project is provided in the following classes (in the case of Java programming language; for C/C++ the logic is similar): `TestCase` (a class with predefined data structures needed to present the input and the output of the algorithms), `AbsAlgorithm` (a class that defines an abstract method that it will be used (when implemented) as the heart of the algorithm) and `TestSetIterator` (a bridge between the definition files and Java data structures; in this class test set configuration file is read and Java test case is generated).

2.2. Types of the ALGATOR engines and users

The ALGATOR was developed to be used as a standalone and/or as a server application. A standalone application is used to develop, test and evaluate algorithms in a separated domain where the results are used only by a limited and typically small group of people. This application can be installed and used on every personal computer. The main drawback of using the standalone version is that the results of the execution can not be fairly compared with the results of other groups of the researchers. On the other hand, the server application offers the possibility to run algorithms provided by different researchers of different groups on a single computer. The results obtained are accurate and comparable. The server version is usually installed on an internet server computer and accessed through the web interfaces.
ALGATOR supports four different user roles: the system administrator (installs and manages the whole system and has the access to all the resources of the system), the project administrator (defines the project and has an access to all the project resources), researcher (defines an algorithm for the selected project, runs predefined tests and compares the results with the results of other algorithms) and the guest (observes all public projects, algorithms, and test results). The logic of the user rights and roles is equally supported in both versions of the ALGATOR, but it is a bit relaxed in the standalone version since all the roles are usually played by a single user.

2.3. The measurements

For each problem there are several different measurements that ensure the correctness and by which one can assess the efficiency of the algorithms. These measurements include the indicators of time consumption and of the quality of the result, counters for the usage frequency of the parts of the program code, and the counters of the usage of the basic execution operations (i.e. the machine instructions). In the ALGATOR system all kind of the measurements are supported and are grouped into three categories: the EM, CNT and JVM indicators.

EM indicators. The EM indicators are used to measure the time and other project-specific metrics . All measurements of the time indicators are performed automatically. To provide as accurate time indicators as possible the ALGATOR tries to reduce the influence of the uncontrolled computer activities (e.g. sudden increase of a system resource usage) by running each algorithm several times. The system measures the first, the best, the worst and the average time of the execution. The project administrator only needs to specify the phases of algorithm execution (e.g. the pre-processing phase, the main phase, the post-processing phase, ...) and to select which of the time indicators are to be presented as the result of execution. The project-specific indicators are defined by the project administrator. They can be presented as a string or as a number. For example, for exact algorithms, the value of an indicator could be ”OK” (if the algorithm produced the correct result) or ”NOK” (if the result of the algorithm is not correct). For approximation algorithms the value of an indicator could be the quality of the result (i.e. the quotient of the correct result and the result of the algorithm).

The values of the EM indicators are generated by ALGATOR performing the following steps:

1. load the test case and create its project-specific representation,
2. load the algorithm (if the algorithm is implemented in Java, for example, the ALGATOR uses the Java reflection capabilities to create the algorithm instance),
3. read the values of the test case specific parameters,
4. run the algorithm and measure the time consumption,
5. read and store the values of the time indicators,
6. determine and store the values of the project-specific indicators,

7. write stored indicators into the output as prescribed in the result description configuration files.

CNT indicators. The CNT indicators (the so called *counters*) are used to count the usage of the parts of the program code. This option is used to analyse the usage of a certain system resource or to count the usage of the selected type of commands on the programming language level. Using this, one can, for example, measure how many times the memory allocation functions were executed during the algorithm execution and the amount of the memory allocated by these calls. One can also use CNT indicators to detect which part of the algorithm is most frequently used. For example, if the problem in concern would be the data-sorting, using the CNT indicators one could count the number of comparisons, the number of swaps of elements and the number of recursive function calls (which are the measures that can predict the algorithm execution behaviour [9]). To facilitate the CNT indicators in the project, the project administrator has to define the names and the meaning of the counters and the researchers have to tag the appropriate places in their code. Everything else is done automatically by the ALGATOR.

JVM indicators. Before the execution of the algorithm, the algorithm code has to be compiled to a code on a lower level. For the C and C++ projects this means that the algorithms are compiled to the machine code of the architecture used by the system, while for the Java projects this means that the algorithms are compiled to the Java bytecode. The performance of the algorithm depends on the number and the type of the low-level instructions used during the execution [5]. The ALGATOR enables the analysis of the low-level instruction usages for the algorithms written in the Java programming language. During the execution of the algorithm ALGATOR counts the bytecode instructions that were used and at the end it prints out the statistics for each instruction (the so called JVM indicators). To enable this facility, a special VMep library [7, 8] was developed and integrated into an open-source Java Virtual Machine JamVM [6]. The VMep enables bytecode counting and makes ALGATOR a very powerful tool for deep analysis of the algorithms' behaviour.
In the rest of this paper we will first present some details about the implementation VMep, then we will give two examples of how the JVM indicators can be used in practice. We will show some method for predicting the time consumption based on the analysis of the JVM indicators.

Note that the EM and the CNT indicators are provided for programs written in Java and C++ language while the JVM indicators are (due to obvious reasons) available only for programs written in Java. Currently, an extension of the system is being developed that will enable machine instruction counting for programs running in a non-virtual environment.

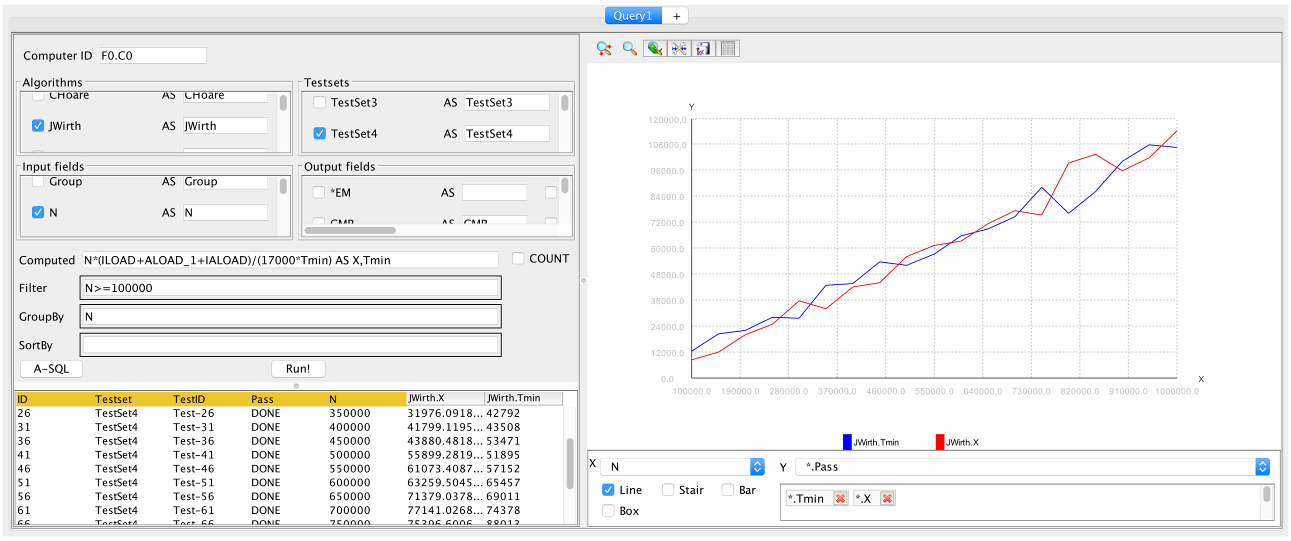
3. USING JAMVM AND VMEP LIBRARY

JamVM [6] is on open source implementation of Java Virtual Machine designed by Robert Lougher. It is a minimal fully operational implementation of JVM written in C and can be translated on several platforms. In order to facilitate Java bytecode counting in JamVM and to simplify the usage of the solution, a special Java library called VMep (Virtual Memory entry point) [7] was developed. To link the system independent Java world with system dependent java virtual machine JamVM, VMep was implemented as a collection of native methods. The main VMep class that supports the initiation and finalisation of the observation is called `Monitor`. It offers methods like `start()`, `stop()` and `addRuntimeFilter()`. The first methods are used to start and to stop (or to pause) the observation, while the latest is used to add filters (i.e. to limit the scope) of the observation and thus to speed up the execution of the program. The `Monitor` class has two important subclasses, namely `InstructionMonitor` and `MemoryMonitor`. The first one is used to provide information about instruction usage while the latest covers the area of memory usage. A usage of VMep library is presented in Listings 2. Here we count the instructions used by the `factorial()` method (using this method the program multiplies the first 100 integers).

```
import jamvm.vmep.InstructionMonitor;
import jamvm.vmep.Opcode;

public class VMepTest {

  static int factorial(int n) {
    int result=1;
    for(int i=2; i<=n; i++){
      result *= i;
    }
    return result;
  }
  public static void main(String[] args) {
    InstructionMonitor monitor = new
        InstructionMonitor();

    monitor.start(); // Start observation
    factorial(100);
    monitor.stop();  // Stop observation

    //
    int[] iUsage = monitor.getCounts();
    for (int i = 0; i < iUsage.length; i++) {
      if (iUsage[i] > 0) {
        String iName = Opcode.getNameFor(i);
        int iFreq    = iUsage[i];
        System.out.printf("%-14s : %d\n", iName,
            iFreq);
      }
    }
  }
}
```

Fig. 2 Using the VMep library to count the instructions usage while multiplying the first 100 integers

The program `VMepTest` from Listings 2 prints a statistics about the instructions usage as presented in Listings 3. We observe that only 16 different Java bytecode instructions (out of 202) were used to calculate the result. The overall number of used instructions was 904 or around 9 instruction per a loop (note that the calculation of



Computer ID F0.C0

Algorithms
- [ ] CHoare AS CHoare
- [✓] JWirth AS JWirth

Testsets
- [ ] TestSet3 AS TestSet3
- [✓] TestSet4 AS TestSet4

Input fields
- [ ] Group AS Group
- [✓] N AS N

Output fields
- [ ] *EM AS
- [ ] CMP AS CMP

Computed N*(ILOAD+ALOAD_1+IALOAD)/(17000*Tmin) AS X,Tmin [ ] COUNT

Filter N>=100000

GroupBy N

SortBy

A-SQL Run!

ID	Testset	TestID	Pass	N	JWirth.X	JWirth.Tmin
26	TestSet4	Test-26	DONE	350000	31976.0918...	42792
31	TestSet4	Test-31	DONE	400000	41799.1195...	43508
36	TestSet4	Test-36	DONE	450000	43880.4818...	53471
41	TestSet4	Test-41	DONE	500000	55899.2819...	51895
46	TestSet4	Test-46	DONE	550000	61073.4087...	57152
51	TestSet4	Test-51	DONE	600000	63259.5045...	65457
56	TestSet4	Test-56	DONE	650000	71379.0378...	69011
61	TestSet4	Test-61	DONE	700000	77141.0268...	74378
66	TestSet4	Test-66	DONE	750000	75306.6006...	88013

X N Y *.Pass

[✓] Line [ ] Stair [ ] Bar [ ] Box

*.Tmin *.X

Fig. 1 Using ALGATOR's analytics module to analyse the Java bytecode usage in Sorting project

`factorial(100)` was performed in a loop that iterated for 99 times). We can also observe that the for-loop condition (performed by `IF_ICMPGT` instruction) is checked for 100-times (which equals the number of loop iterations plus one).

```
ICONST_1       : 1
ICONST_2       : 1
BIPUSH         : 1
ILOAD_0        : 100
ILOAD_1        : 100
ILOAD_2        : 199
ALOAD_0        : 1
ALOAD_1        : 1
ISTORE_1       : 100
ISTORE_2       : 1
IMUL           : 99
IINC           : 99
IF_ICMPGT      : 100
GOTO           : 99
IRETURN        : 1
INVOKEVIRTUAL  : 1
```

Fig. 3 A result produced by the `VMepTest` class

Due to additional tasks that are performed by the VMEP library during the execution of algorithms (i.e. collecting the information on instructions and memory usage), the running times of the algorithms executed by JamVM are significantly bigger than those obtained by running the same algorithms on a standard VM. The slowdown factor tends to be a constant, around 3.6 on average.

VMep library in integrated in ALGator in such a way that a final user can use it without actually knowing its implementation details. By calling the appropriate ALGATOR's module, the user gets Java bytecode statistics ready to be analysed by the analytics module. More precisely, by executing, for example, `java algator.Execute Sorting -m jvm`, ALGATOR will run all algorithms in the `Sorting` project and save the results to output files. By calling `java algator.Analyse Sorting` the user will be able to analyse the results and produce tables, charts and graphs as depicted in Figure 1. In this example we were looking for a relation between the minimal execution time (`Tmin`) and the three most frequently used Java bytecode instructions (i.e., `ILOAD`, `ALOAD_1`, and `IALOAD`) in the `JWirth` algorithm. We found out that `Tmin` (the blue line in graph) very closely correlates with `X` (the red line in graph), where `X` was defined to be

$$X = \frac{N * (ILOAD + ALOAD\_1 + IALOAD)}{17000 * Tmin}. \qquad (1)$$

4. JVM INDICATORS IN PRACTICE

To explore the measuring capabilities of the ALGATOR we chose a simple matrix multiplication problem: given two square matrices A and B each containing n^2 elements (a_{ij} and b_{ij} for $i, j = 0, \ldots, n-1$) calculate the elements of a square matrix C by

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}.$$

Since the number of operations in this formula is cubic to n, we can reasonably expect that the time complexity of any algorithm implementing this formula would have the time complexity $\Theta(n^3)$. The simple implementation of this formula is presented in listings in Figure 4.

```
void MUL(int[][] A, int[][] B, int[][] C) {
  for (int i = 0; i < A.length; i++) {
    for (int j = 0; j < A.length; j++) {
      for (int k = 0; k < A.length; k++) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Fig. 4 The Java code for the `MUL` algorithm

We named this implementation the `MUL` algorithm since its main (and the most consumptive) operation is the multiplication. Using the ALGATOR's time complexity indicators we measured the time needed to execute this algorithm on a set of test cases with dimensions n ranging from 200

n	ICONST_0	ILOAD	ALOAD_1	ALOAD_2	ALOAD_3	IALOAD	AALOAD	ISTORE
10	111	7221	2221	1000	1000	3000	3000	111
20	421	56841	16841	8000	8000	24000	24000	421
30	931	190861	55861	27000	27000	81000	81000	931
40	1641	451281	131281	64000	64000	192000	192000	1641
50	2551	880101	255101	125000	125000	375000	375000	2551

	IASTORE	DUP2	IADD	IMUL	IINC	IF_ICMPGE	GOTO	ARRAYLENGTH
10	1000	1000	1000	1000	1110	1221	1110	1221
20	8000	8000	8000	8000	8420	8841	8420	8841
30	27000	27000	27000	27000	27930	28861	27930	28861
40	64000	64000	64000	64000	65640	67281	65640	67281
50	125000	125000	125000	125000	127550	130101	127550	130101

Table 1 The Java bytecode instructions usages in the matrix multiplication algorithm.

to 500. We run all the tests described in this paper on a personal computer with Intel(R) Core(TM) i7-6700 CPU running at 3.40GHz with 32Gb of memory. The execution of the matrix multiplication for smaller inputs (n=200) was done in 6000 microseconds and for larger matrices (n=500) in 140.000 microseconds. To eliminate the impact of the real environment we executed all the tests (i.e. we calculated each product) for 500 times and we took the minimal time of all executions (obviously, this is the time in which the execution can be performed if the environmental influences are as small as possible). The time of the execution for the matrix multiplication problem is depicted in Figure 6 with blue line.

```
void MUL(int[][], int[][], int[][]);
   0: iconst_0        |    36: dup2
   1: istore 4        |    37: iaload
   3: iload 4         |    38: aload_1
   5: aload_1         |    39: iload 4
   6: arraylength     |    41: aaload
   7: if_icmpge 73    |    42: iload 6
  10: iconst_0        |    44: iaload
  11: istore 5        |    45: aload_2
  13: iload 5         |    46: iload 6
  15: aload_1         |    48: aaload
  16: arraylength     |    49: iload 5
  17: if_icmpge 67    |    51: iaload
  20: iconst_0        |    52: imul
  21: istore 6        |    53: iadd
  23: iload 6         |    54: iastore
  25: aload_1         |    55: iinc 6, 1
  26: arraylength     |    58: goto 23
  27: if_icmpge 61    |    61: iinc 5, 1
  30: aload_3         |    64: goto 13
  31: iload 4         |    67: iinc 4, 1
  33: aaload          |    70: goto 3
  34: iload 5         |    73: return
```

Fig. 5 The Java bytecode for the `MUL` algorithm

In the same graph a simple prediction for the execution time is also depicted. It was calculated by a simple method called `Calc1`. In this method we calculated a multiplication factor $c = avg(time_i/n^3)$. The red dots in Figure 6 represents a graph of a function cn^3. Obviously the red dots are of the same shape as the blue line (which is due to the fact that our algorithm has $\Theta(n^3)$ time complexity) but it is not very accurate. An average error (i.e. the difference between measured (blue) and calculated (red) time divided by measured time) is 11,25%. This error is a bit smaller (i.e 7,1%) if we take only bigger dimensions of input matrices (from 300 to 500), but it is still relatively big. Therefore the method `Calc1` can not be considered a very successful method.

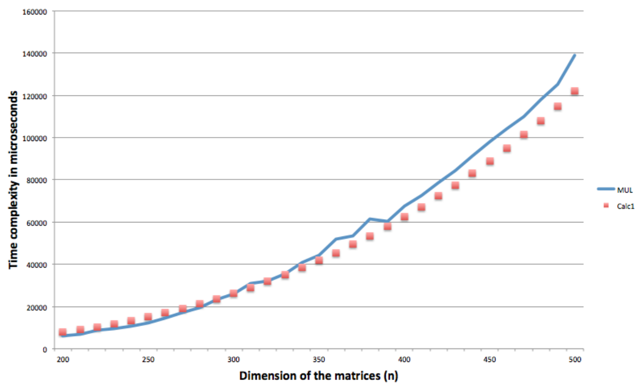


Fig. 6 The time complexity of the `MUL` algorithm (blue line) and the performance prediction calculated by a simple `Calc1` method (red dots).

In order to find better performance prediction for this algorithm we used ALGATOR's capability for measuring the usages of the Java bytecode instructions (the bytecode for the algorithm `MUL` is listed in Figure 5). The results show that only 16 (out of 202) Java bytecode instructions are used during the execution of this algorithm: 10 instructions for the stack manipulation (ICONST_0, ILOAD, ALOAD_1, ALOAD_2, ALOAD_3, IALOAD, AALOAD, ISTORE, IASTORE, DUP2), two instructions to control the flow of the program (IF_ICMPGE, GOTO), the ARRAYLENGTH instruction used to determine the size of an array, and three arithmetic instructions (IADD, IMUL, IINC). The frequencies of the usages of these instructions for the matrices of sizes from 10 to 50 are presented in Table 1. As it is clearly seen from the data in the table, for most of the instructions their usages in the matrix multiplication algorithm is of the order $\Theta(n^3)$. The only exceptions are the instructions ICONST_0 and ISTORE with the order $\Theta(n^2)$. From the data presented in Table 1 we calculated the overall number of the instructions $\mathtt{INST}(n)$ used in the `MUL` algorithm:

$$\mathtt{INST}(n) = 25n^3 + 12n^2 + 12n + 6.$$

This means that in the case of $n = 500$, for example, the JVM performs $25 \times 500^3 + 12 \times 500^2 + 12 \times 500 + 6 = 3.128.006.006$ bytecode instructions to execute the MUL algorithm. Since this execution requires approximately 140.000 microseconds, an average time to execute one Java byte code instructions is 0,044ns.
Analyzing the results presented in Table 1 (extended with measurements for n=60, ..., 500) a natural question arises: can we calculate an average time (over all the measurements) used to execute one bytecode instruction and use this average to predict the behaviour (i.e. time consumption) of the MUL algorithm for a given n. To find an answer to this question, we propose the following method Calc2: calculate the average time I_n used for one bytecode instructions while performing MUL on the matrix of size n (e.g. $I_{500} = 0,044$) and calculate I as an average of I_n. Then use I to estimate the execution time of MUL by $T(n) = I * INST(n)$. Using this method we calculated $I = 0,039ns$ (note that we used only measurements for $n = 300, ..., 500$ since we assume that the measured times are much more accurate for bigger inputs). Surprisingly, the Calc2 method gives very similar results as the method Calc1: an average difference between those two methods is 0,03% for $n = 300, \ldots, 500$. In other words, calculating the uniform average time per bytecode instruction yields another useless method for estimation of time consumption.

The main reason for bad results is that some bytecode instructions are much more expensive than the others. For example, we can reasonably assume that the IMUL instructions takes much more time to execute than the ILOAD instruction (the first instruction multiplies two integers while the second one loads an integer onto a stack). The question is, how many different types of instructions (instructions of the same type take approximately the same time to execute) are included in the MUL algorithm. To answer this question we implemented two algorithms, both of then very similar to MUL. The first one, the ADD algorithm, is an exact copy of MUL with the only difference in the line 5 where we instead of multiplication use addition (C[i][j] += A[i][k] + B[k][j];). The execution of this algorithm results in the usages of exactly the same Java bytecode instructions, the only difference is that instead of IMUL in ADD algorithm only IADD instruction is used (which is logical, but we also proved this by scanning the ALGATOR's jvm indicators). As a consequence, in the MUL we have n^3 IADDs and n^3 IMULs while in ADD we have only $2 \times n^3$ IADDs. The number of all the other instructions is equal in both algorithms. In the second algorithm, SET, we deleted line 5 of MUL and replaced it with 4 lines as showed in the listings in Figure 7.

```
void SET(int [][] A, B, C) {
  int x=0,y;
  for (int i = 0; i < A.length; i++) {
    for (int j = 0; j < A.length; j++) {
      for (int k = 0; k < A.length; k++) {
        y = A[i][j];
        B[i][j] = x;
        C[i][j] = y;
        x++;
      }
    }
  }
}
```

Fig. 7 The Java code for the SET algorithm

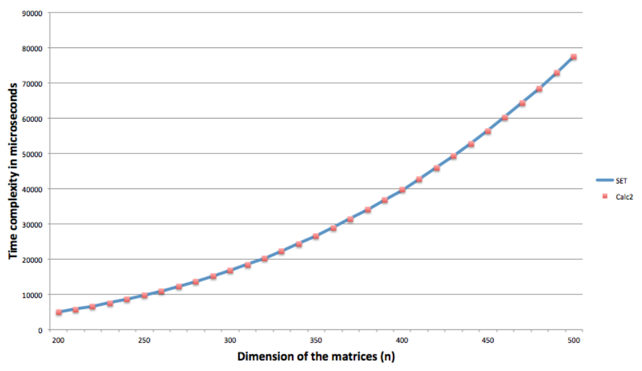


Fig. 8 The time complexity of the SET algorithm (blue line) and performance prediction calculated by a Calc2 method (red dots).

The resulting SET algorithm compiles into a Java bytecode program with exactly the same number of instructions as the MUL, which means that for executing SET on matrices of size n, JVM also performs $INST(n)$ bytecode instructions. The only difference is that the SET does not use the IADD and IMUL instructions.

The algorithm SET uses only the following twelve instructions: ICONST_0, ILOAD, ALOAD_1, ALOAD_2, ALOAD_3, IALOAD, AALOAD, ISTORE, IINC, IF_ICMPGE, GOTO, ARRAYLENGTH. We made an assumption that these instructions are all equally consumptive, we named them as "simple instructions", and we used the method Calc2 to calculate their average execution time I. Using the resulting $I = 0,0248ns$ (for the further reference we will denote it with I_s) and formula $T_{SET}(n) = I_S * INST(n)$ we found out that the Calc2 methods in this case yields almost a perfect estimation. Figure 8 shows the measured time of the SET method (blue line) and its estimation provided by Calc2 method. An average error ($n = 200, \ldots\ldots, 500$) of this method is 0,4%. This means that the calculated $I_s = 0,0248$ nanoseconds is a reasonably good estimation for the execution time of every simple Java bytecode instruction on this computer.

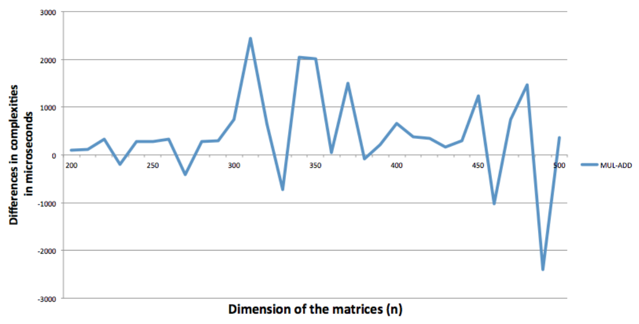


Fig. 9 The differences in the time complexities of the MUL and ADD algorithms.

To make a good estimation for the MUL algorithm we now only have to determine the estimation for the time complexities of the IMUL and IADD instructions. First we compare the execution time of the algorithms MUL and ADD to find out that these two algorithms are comparable in the sense of time consumption. Graph 9 shows the differences in the time complexities of the `MUL` and `ADD` algorithms. According to the oscillation of the graph we can conclude that both algorithms are equally consumptive, the repetitive exchange of the leadership (positive and negative values on the graph) indicates that the times of execution were measured with a certain relatively small (on average less than 1%) error. Since the only difference between `MUL` and `ADD` is in the number of IMUL and IADD instructions used (the second one uses only IADDs while the first one uses both) and since we proved that there is no real difference in time consumption, we can conclude that the time consumptions of the IADD and IMUL instructions are the same. This is not just an interesting result but it also gives us an opportunity to estimate the real time consumption of both arithmetic instructions. In the `MUL` we have $INST(n)$ instructions among which there are $2 \times n^3$ arithmetic instructions. Assuming that the time complexity of an arithmetic instruction (I_A) equals $I_A = I_S + \lambda$ we get

$$\lambda = AVG_n\left(\frac{T_{MUL}(n) - T_{SET}(n)}{2 * n^3}\right).$$

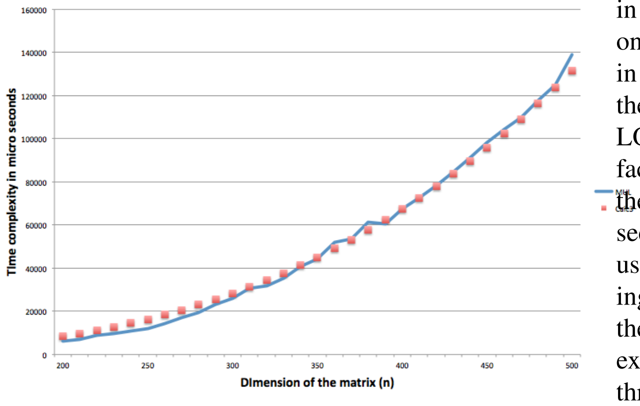


Fig. 10 The time complexity of the `MUL` algorithm (blue line) and performance prediction calculated by a `Calc3` method (red dots).

Using the measured times of `MUL` and `SET` and averaging for $n = 300, \ldots, 500$ we obtain $\lambda = 0,22$ and $I_A = 0,24$ nanoseconds. This means that an average cost of an arithmetic operations IADD and IMUL is 9,7-times bigger than an average cost of a simple instruction.

To estimate the execution time of the `MUL` algorithm we use the following `Calc3` method: given the factors I_S and I_A, calculate the estimation of the time complexity of the `MUL` algorithm by

$$T_{MUL}(n) = (INST(n) - 2n^3) \times I_S + 2n^3 \times I_A.$$

Using this estimation we find out that it much better fits the `MUL` algorithm then the previous ones. Graph in Figure 10 shows the time complexity of `MUL` with blue line and the `Calc3` estimation with red dots. An average error of this estimation (for $n = 300, \ldots, 500$) is 2.3%.

For the next example of JVM indicators usage let us consider the data-sorting problem. Here an algorithm aims to sort (in a prescribed order) the input array of numbers. It is well known that the fast sorting algorithms can perform this task in $\mathcal{O}(n \log n)$ time. In our experiment we used the so-called Wirth's algorithm, which is a special case of the QuickSort [9] sorting algorithms. It uses one pivot (the first element of the input array) to split the array into two sub-arrays (one with numbers that are less than (or equal to) the pivot and the other with numbers that are greater then (or equal to) the pivot) and then it sorts these arrays recursively (see algorithm's code in Listings 12). Running this algorithm in ALGATOR reveals that the Java bytecode of this algorithm uses only 17 different instructions, namely, `IFGT`, `ISUB`, `ALOAD_0`, `INVOKEVIRTUAL`, `RETURN`, `ILOAD_3`, `ILOAD_2`, `IF_ICMPGT`, `ISTORE`, `IASTORE`, `IF_ICMPGE`, `IF_ICMPLE`, `GOTO`, `IINC`, `IALOAD`, `ALOAD_1`, and `ILOAD`. It is interesting (but according tho the nature of the Java virtual machine, which is a stack oriented machine, not very surprising) that a majority of work is done by only three instructions: `IALOAD`, `ALOAD_1`, and `ILOAD`. The number of all instructions used by Wirth's algorithm when sorting arrays of sizes from 100000 to 500000 if presented Table 2. In this table the number of the three LOAD instructions are presented in the first line, the number of all instructions in the second and the quotient between the first and the second value in the third. For the test cases used in this experiment all the quotients were about 0.60, which means that the three LOAD instructions perform about 60% of all work. This fact can also be observed in a graph in Figure 11 where the number of each instruction used is depicted. We can see that most instructions are used very rarely, some are used moderately and a few of them very frequently. Using this observation and concrete numbers obtained with the experiment one could derived a formula to predict the execution time being dependant only on the number of the three LOAD instructions used while executing the algorithm (see Formula 1).

```
void wirth(int[] a, int l, int r) {
  int left = l, right = r;

  if (right - left <= 0) return;

  int pivot = a[l];
  while (left <= right) {
    while (a[left]  < pivot) left++;
    while (a[right] > pivot) right--;

    if (left > right) break;

    int tmp = a[right];
    a[right--] = a[left];
    a[left++] = tmp;
  }
  wirth(a, l, right);
  wirth(a, left, r);
}
```

Fig. 12 The Java code for the Wirth's QuickSort algorithm

5. CONCLUSIONS

In this paper we described the ALGATOR – a system for testing and analysing the algorithms. We showed how

N	100000	150000	200000	250000	300000	350000	400000	450000	500000
SUM: IALOAD + ALOAD_1 + ILOAD	17128368	26971644	36880192	46903030	55383357	66461302	77290334	88639256	98630370
ALL instructions	28469965	44810573	61265870	77790644	91612964	110011072	128043459	146915394	163262811
SUM /ALL	0,6016294	0,60190357	0,60196961	0,60293922	0,60453624	0,60413285	0,60362579	0,60333539	0,604120249

Table 2 Usage of Java bytecode instructions for Wirth's algorithm.

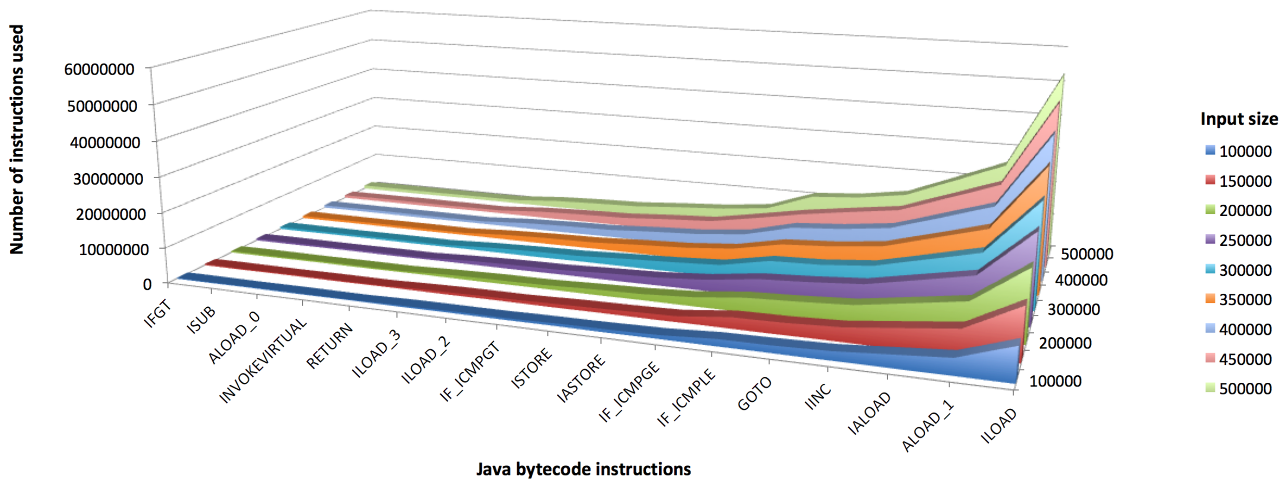


Fig. 11 A statistics of the JVM instructions usage

to use the ALGATOR's ability to count the usages of the Java bytecode instructions. Using three algorithms (`MUL`, `ADD` and `SET`) we presented different (more or less efficient) methods to produce the performance prediction of the algorithms based on the number of Java bytecode instruction used. We showed that the "simple" instructions (e.g. ILOAD_0, IALOAD, ISTORE, ...) are equally time consumptive and that they on average take 0,0248 nano seconds to execute (on our computer). We also showed that arithmetic instructions (IADD and IMUL) are much more time consumptive - on our computer these instructions take 0,24 nano seconds (which is almost 10 times slower than the simple instructions). Using these information about bytecode instructions and the formula for total bytecode instruction usages (which was also derived from the results of ALGATOR's execution) we presented a method for the execution time prediction of the selected algorithm for matrix multiplication. The results of this method were much better then the results of a basic (naive) method which estimates the time complexity with a simple cubic function.

The ALGATOR's capability to count the Java bytecode usages helps us to better understand the behaviour of the algorithms. The test case presented in this paper is very educative, but it is not general because of the nature of the selected algorithm (the behaviour of the algorithm is totally deterministic and does not dependent on the input data; the algorithm always uses the same instructions regardless the content of the input matrices). To prove a general usability of the JVM indicators other problems and algorithms should be concerned.

REFERENCES

[1] T. DOBRAVEC. ALGator - an open source automatic algorithm evaluation system. https://github.com/ALGatorDevel/Algator, 2012-2018.

[2] T. DOBRAVEC. Estimating the time complexity of the algorithms by counting the java bytecode instructions. In *Informatics 2017, Poprad, Slovakia*, pages 74–79, November 2017.

[3] S. F. GOLDSMITH, A. S. AIKEN, and D. S. WILKERSON. Measuring empirical computational complexity. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404, September 2007.

[4] H. H. HOOS. A bootstrap approach to analysing the scaling of empirical run-time data with problem size. Technical Report TR-2009-16, University of British Columbia, 2009.

[5] J. M. LAMBERT and J. F. POWER. Platform independent timing of java virtual machine bytecode instructions. *Electronic Notes in Theoretical Computer Science*, 220:79–113, 2008.

[6] R. LOUGHER. JamVM - an open source Java virtual machine. jamvm.sourceforge.net, 2014.

[7] J. NIKOLAJ. Java virtual machine for counting the Java bytecode usage (original title: Predelava javanskega navideznega stroja za štetje ukazov zložne kode, language: Slovene). *University of Ljubljana, Faculty of Computer and Information Science*, 2014.

[8] J. NIKOLAJ. A source code of VMEP virtual machine. github.com/nikolai5slo/jamvm, 2014.

[9] R. SEGEDWICK. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1977.

Received May 10, 2018, accepted November 27, 2018

BIOGRAPHY

Tomaž Dobravec received his Dipl. Ing. degree (1996) in mathematical science and his Ph.D. (2004) in computer science, both from the University of Ljubljana, Slovenia. He is an Assistant Professor at the Faculty of Computer and Information Science, University of Ljubljana. His main research interests are in algorithm design, analyses and evaluation, in theory of programming languages and in networks.