International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:3, No:1, 2009

FHOJ: a new Java benchmark framework

Vinh Quang La
International Business Consulting
University of Applied Sciences Offenburg, Germany
Email: vla@stud.fh-offenburg.de
Dirk Jansen
Department of Electrical Engineering
University of Applied Sciences Offenburg, Germany
Email: d.jansen@fh-offenburg.de

Abstract—There are some existing Java benchmarks, application benchmarks as well as micro benchmarks or mixture both of them, such as: Java Grande, Spec98, CaffeMark, HBech, etc. But none of them deal with behaviors of multi tasks operating systems. As a result, the achieved outputs are not satisfied for performance evaluation engineers.

Behaviors of multi tasks operating systems are based on a schedule management which is employed in these systems. Different processes can have different priority to share the same resources. The time is measured by estimating from applications started to it is finished does not reflect the real time value which the system need for running those programs. New approach to this problem should be done.

Having said that, in this paper we present a new Java benchmark, named FHOJ benchmark, which directly deals with multi tasks behaviors of a system. Our study shows that in some cases, results from FHOJ benchmark are far more reliable in comparison with some existing Java benchmarks.

Keywords—Java Virtual Machine, Java benchmark, FHOJ framework.

I. Introduction

Because of the dynamic linking mechanism, it makes the Java benchmark more difficult to implement in comparison with C/C++ benchmark. In C/C++ benchmark we can tell about the performance of a system without even running it, through seeing binary code which is achieved after the linking process. Since the linking process in Java is dynamic, performance factors when a JVM interacts with the class file library must be added on into a Java benchmark.

In the context of multi thead systems, behavior of the system is not linear to different tasks that are in use at the same time. Those differences are based on the different employed scheduling mechanism in this system. Generally, the linear time measurement method in Java for a single task in a multi task system does not produce precise values.

The linear time measurement is the way to use the timer, eg: *System.currentTimeMillis()* or *System.nanoTime()* which Java runtime environment supplies. In most cases, Java runtime environment encapsulates Timer which is supported by the OS with some modifications in its features. Virtually, it adds more burden to the system since more processing time is needed for object timers instantiation. That hides the real value of time spent on each single process. To our knowledge, there is currently no Java runtime environment that provides functions to access real values of the run time that a system spends for programs.

In the FHOJ benchmark, instead of using the timer operations which Java runtime environment supplies, we use a native timer operations that OS supplies. Therefore, we can separate the time the system spends for its permanent services and life time of a program, thus actual run time of a program is achieved.

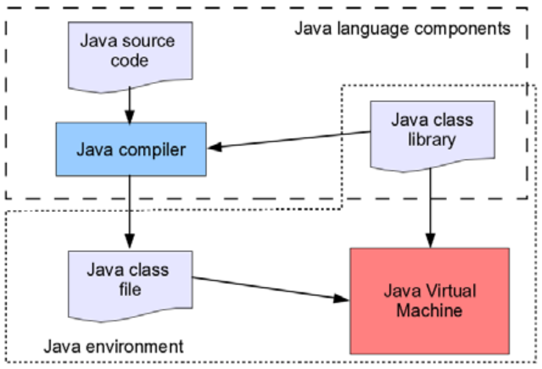


Fig. 1. Components in Java technology architecture

The remaining parts of this paper are structured as follows: section 2 introduces technological background of this work and discussions about related works. Section 3 presents the benchmark setup. Section 4 provides the results of this benchmark and some discussions. Finally, section 5, presents our conclusions.

II. Background and Related work

A. Linear time measurement

Most current Operating Systems (OS) are multi task systems. There are many jobs can be served at the same time. To fulfil this characteristic a typical OS has a superior mode (called kernel mode in Linux), which has the highest priority to do continuous pooling to distribute and revoke resources. The superior mode or kernel mode normally can not be interfered by any lower priority modes in which the applications are executed.

However, a typical way to measure the runtime of a program is to use the timer provided by the Java runtime environment provides. It is shown as follows:

```
long tStart = 0;
long tEnd = 0;

tStart = System.currentTimeMillis();

   // some business code here

tEnd = System.currentTimeMillis();
System.out.println(tStart - tEnd);
```

In between two timer calls, *System.currentTimeMillis()*, it can be anything. It can be a whole Java benchmark workload or just only partial of a benchmark workload, it is depended on different implementers. The respective Java bytecodes of the Java program above is shown below.

```
1:   lconst_0
2:   lstore_1
3:   lconst_0
4:   lstore_3
5:   invokestatic    #2; /invoke timer
6:   lstore_1
7:   invokestatic    #2; /invoke timer
8:   lstore_3
9:   getstatic       #3;
10:  lload_3
11:  lload_1
12:  lsub                /get difference
13:  invokevirtual   #4;
```

The method *System.currentTimeMillis()* is translated into bytecodes at line 5 and line 7 as *invokestatic* and its argument is value Nr.2 to the constant pool of the class file. These *invokestatic* call a static method as a timer resolutions. The runtime of a program is calculated as the difference between the values of the two timers through the *sub* at line 12.

At the system level, when the line *invokestatic #2* is executed then a native function *clock_get_time* is called. The return values are the difference, measured in milliseconds, between the actual current time and midnight, January 1, 1970 UTC.

This method is called the linear time measurement because the time spending for the running services and other existing programs in the system can add up to the results. As illustrated in Figure 2, the actual time needed for Task 1 is: $\Delta_{t1} + \Delta_{t2} + \Delta_{t3}$.

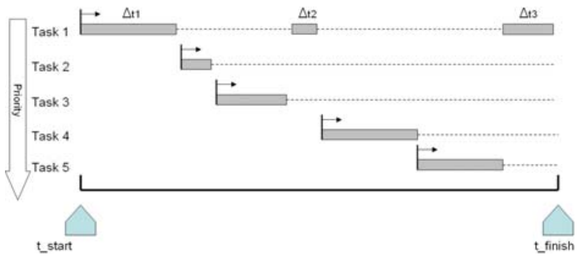


Fig. 2. Jobs are executed in a multi tasks Operating System

B. User time and System time

Any application has to go through at least two modes: kernel mode and user mode, the running time respectively to them are system time and user time, see Figure 3. The starting point and the ending point of a task are done by the kernel mode. During the execution time in user mode, an application may require the kernel mode to provide resources to fullfill its duty. For instance: a typing program in use to write this paper has to switch back and forth both application mode and kernel mode whenever the ”Save” button is clicked.

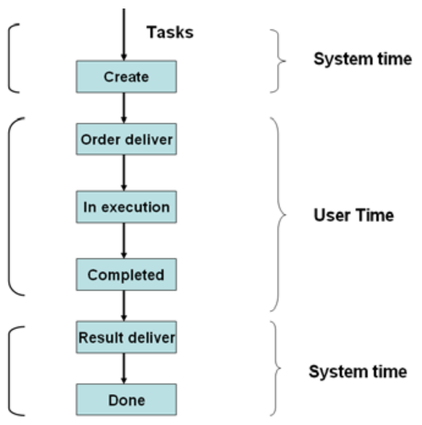


Fig. 3. How time has gone when tasks are executed in a multi tasks system

To have a clear distinguish between user time and system is very important in a Java micro benchmark. Because of the dynamic linking process of the JVM, sometimes the loading time for required parts of the system class library is more consuming than time for executing tasks. However, to separate user time and system time is the duty of an accounting system of a OS. That means if a OS do not support this features then its kernel scheaduler must be rewrited.

The dynamic linking mechanism in Java is happened when the program is executed, the JVM loads program classes and interfaces and hooks them together. This dynamic linking mechanism is totally different to the traditional linking in C++. Figure 4 and 5 shows those difference. In Java technology one more step has to be made to load Java library, instead of taking directly binary code from the dynamic library (*.dll in Windows or *.so in Unix). This feature also makes the question which places to put the timers is very important for any benchmark.

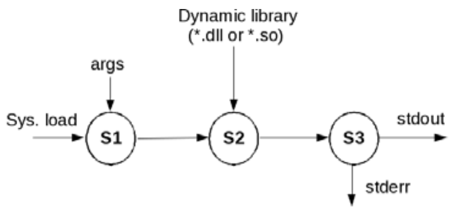


Fig. 4. State transitions in a dynamic linking system

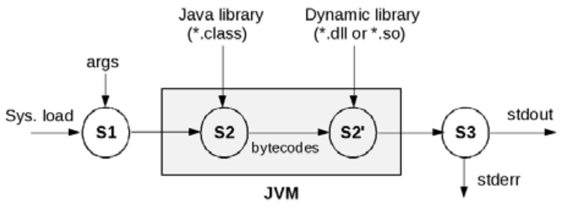


Fig. 5. Dynamic linking states in Java architecture

Since the running time of a Java macro benchmark are normally much more bigger than its system time. That is the reasons in most cases, the combination values of user time and system time of an application is considered as its running time.

C. Timer service implementation

There are two ways to implemente timer services in a system: soft-timer and hard-timer. Soft-timers are preferred to hard timers in a general purpose OS because they reduce the overhead of time consuming task of interrupts. Normally, a OS supplies the timer and its according operations can categories into 3 API levels:

- Level 1: provides low-level hardware related operations
- Level 2: provides soft-timer related services
- Level 3: provides access either to the storage of the real-time clock or to the system clock

The first level of operations, considered as low-level system operations, is developed and provided by board support package developers. The second level of timer-related operations includes the core timer operations that are heavily used by both the system modules and applications. Either an independent timer-handling facility or a built-in one that is part of the kernel offers these operations. The third level operations are mainly used by user-level applications. The operations in this level interact either with the system clock or with the real-time clock. Its typical operations are *clock_get_time* and *clock_set_time*.

Figure 6 depicts steps in servicing the timer interrupt. The soft-timer facility is responsible for maintaining soft timers

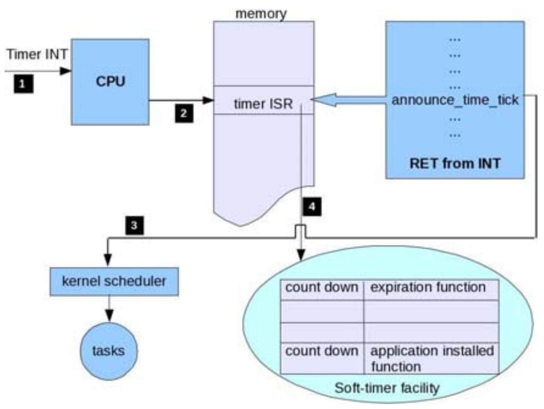


Fig. 6. Steps in servicing the timer interrupt [1]

at each timer tick. It is comprised of two components: one component lives within the timer tick ISR and the other component lives in the context of a task. This feature helps to keep the system clock is not drift away if the task spans in multiple ticks.

Note that the clock resolutions are system-dependent and can not be set by the JVM implementer. While the unit of time of the return value is a millisecond, the granularity of the values depends on the underlying operating system and may be larger. In the Linux, the resolution is usually 10ms on 32-bit architectures and 1ms on 64-bit architectures.

D. Runtime and loading time

The runtime or execution time of a Java program is shown in the equation below:

Runtime = start up time + translation time + bytecode execution time

Start up time is the time a JVM needed to start its internal data structures and invoke its components. In some JVMs, the Java program and the input to this program are loaded into the memory. Based on its runtime library the execution path of the program can be built in the linking processe. All of these steps are happened in the start up time, ready for the second step, which is the translation step.

The translation time is the time spent on translating bytecodes, from the standard bytecodes, the bycodes defined by Sun, to internal built-in specific purpose bytecodes of a JVM. The translation varies with different JVMs. Typically, a JVM with an interpreter architecture spends no time for translating. While other JVMs, such as the HotSpot, heavily optimize certain bytecodes, and translating bytecodes more than once; which results in an increased translation time.

The bytecode execution time of the JVM depends on how many bytecodes need to be executed in a code path. And the code path can be changed accordingly to the input of the program. A visual example we can give is a graphical user interface application (GUI), whenever an user triggers any GUI element then a code path is built accordingly to the event.

In general, by using current native timer functions at Level 2 of a OS we can find the runtime of a program precisely, as we did in FHOJ benchmark framework. Nevertheless, to caculate start up time and bytecode execution time of a JVM, a high resolution timers [2] and a fine tuning system accounting must be built. It is of interest to this reseach.

To make comparision in runtime of Java program between JVMs, the standard deviation σ is used to show the spread of the values of measure times away from its mean. The lower deviation, the better results of a benchmark are.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2} \qquad (1)$$

Where N is the number of times to run a program, $\bar{x}$ is the mean of x_i, caculated by the equation below.

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad (2)$$

E. Related work

Java Grande [3], [4] consists of benchmarks in three categories: low-level operations, kernel and large scale application. The low-level operations catergory measures the performance of JVM such as: arithmetic and math library operations, garbage collection and method calls and casting. The kernel category consists of small programs, including Fourier coefficient analysis, LU factorization, alphabeta pruned search, heap sort and IDEA. The large scale application is intended to represent real application programs, currently there is only one application in this catergory.

Caffemark [5] has been developed to measure some aspects of Java such as: loop performance, the execution of decisions-marking instructions and recursive function calls. However, the Caffemark Benchmark does not emulate the operations of real-world programs.

Spec98 [6] was developed with the expectation of being the standard for performance evaluation of JVM. It measures the efficiency of JVM, the just-in-time (JIT) compiler, and operating system implementations. On the hardware side, it includes CPU (integer and floating-point), cache, memory, and other platform-specific performance.

HBench Java [7] introduces vectors based on the method to characterize a JVM in an abstract level. The principle of this Benchmark is the observation that a system's performance is determined by the performance of individual primitive operations that support. By introducing a simple linear model rather than a complex model, HBench retain the simplicity of a linear model by adding multiple data points for single primitives. The basic primitive operation of this model is the JVM's assembly instructions or bytecode. The higher abstract levels are: system classes, memory management, execution engine and JIT.

Linpack Benchmark [8] is a measure of a system's floating point rate of execution. It is determined by running a computer program that solves a dense system of a linear equation, which is a common task in engineering. The solution is obtained by Gaussian elimination with partial pivoting, the result is reported in millions of floating point operations per second (Mflop/s).

PennBench [9] is another Java benchmark, it is claimed to be the first Java benchmark for the embedded system especially to cell-phone and PDA like devices. This Benchmark focuses mainly on the memory characteristic of a JVM according to its input applications.

The DaCapo benchmark suite [10], [11] is a set of 11 non-trivial, real-world, open source Java benchmarks, intended as a tool for Java benchmarking by the programming language, memory management and computer architecture communities. It is supposed that it superiors than SPEC Java in a variety of way, including more complex code, richer object behaviors, and more demanding memory system requirements.

Although each of the benchmarks mentioned above was designed for a specific purpose. However, there is one thing in common in all mentioned benchmarks that is all of them use the linear time measurement when they want to measure running time of applications. This is a common weakness of all benchmarks.

III. FHOJ BENCHMARK FRAMEWORK

The purpose of this benchmark is mainly to measure the performance of the JVM. However, it also covers a part of job of the compiler step, changing file header of workload in order to make them compatible to all JVMs, see Figure: 7. The Testset or the workload, consists of a bundle of Java programs which have to be passed through Java compiler to get binary files (*.class) as the input for the benchmark. Then those files are triggered to run by the BenchmarkCtr module.

The whole BenchmarkCtr module is written in shell script, makes the use of API timer functions at the level 2 of timer operations the OS supply. It is designed to compatible with Unix-like and Solaris systems. The BenchmarkCtr is responsible for tuning rules in the FHOJ benchmark, eg: number of execution times, input parameters, mean value calculation, etc.

The numbers of applications in the Testset can be changed depending on the specific purposes to do the benchmark. The workload of other benchmarks can be also directed into the FHOJ framework. In this case what we have to do is simply to eliminate the timers functions and the output result in those benchmark.

Log files will be generated after the execution process. Then reports will be created base on the Log files of the JVMs. Up till now, all information gather from Log files for creating reports must be done manually, such as: mean value of execution time and graphics generation, further improvement will be done later.

IV. BENCHMARK SETUP AND RESULTS

A. Benchmark setup

The benchmark is setup with 5 Java virtual machines: SUN JVM 1.4, SUN JVM 1.6 Client and Server version, IBM JVM 1.4 and KVM. The Client and Server version of SUN JVM

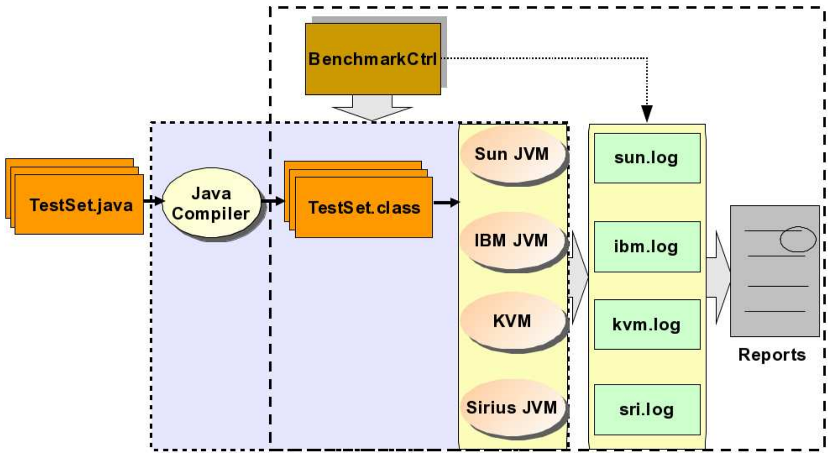


Fig. 7. FHOJ Benchmark framework

Name of JVM	Size (MB)	Number of class
SUN JVM 1.6	41.708	16482
SUN JVM 1.4	26.311	9435
IBM JVM 1.4	22.152	15075
KVM/CLDC	0.179	116

TABLE I
THE SIZE OF JAVA CLASS LIBRARIES IN THIS BENCHMARK

1.6 are set by changing the parameters of the SUN JVM 1.6. No changes had been made in all the runtime lib (JRL) which accompany with those JVMs. SUN JVM 1.6 Client and Server version employ the same JRL. Table I shows the size of the JRL used in this benchmark.

The KVM uses the Interpreter execution engine in which every single Java bytecode has to be translated into one or many native codes step by step. The SUN JVM 1.4.2 and IBM JVM 1.4.2 have Just In Time (JIT) architecture which is considered the second generation of a JVM. The SUN JVM 1.6.0 uses Dynamic Adaptive Compilation (DAC) architecture, the JIT with some improvements in Hotspot mechanism.

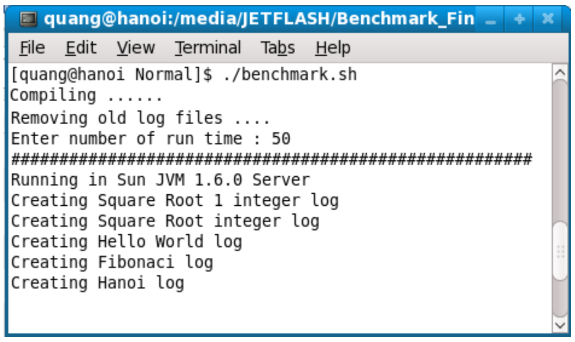
As mentioned above, the workload of other benchmark can be redirected to the FHOJ framework if it is needed. In the scope of this study, our workload consists of just only 18 Java programs, covers 4 areas of features of Java programming language:

- Simple arithmetic computation
- Large computation
- Large computation with multiple stacks called
- Object oriented and polymorphism

B. Results and discussion

Firgure 8 shows the user interface of the FHOJ benchmark. In the following sections, we present some achieved results in the scope of this study.

1) Precompiled system class with ROMizing technique: There is no answer for the question which JVM is the best. One JVM can be very good at class loading part but it can perform poorly at garbage collection or at execution engine,


quang@hanoi:/media/JETFLASH/Benchmark_Fin
File Edit View Terminal Tabs Help

```
[quang@hanoi Normal]$ ./benchmark.sh
Compiling ......
Removing old log files ....
Enter number of run time : 50
#####################################################
Running in Sun JVM 1.6.0 Server
Creating Square Root 1 integer log
Creating Square Root integer log
Creating Hello World log
Creating Fibonaci log
Creating Hanoi log
```

Fig. 8. The user interface of the FHOJ

etc. That is the reason why we have variant answers for different programs, see Figure 9. For the most programs in the benchmark Sun JVM 1.6.0 Server is the best but it is come third for the *matrix* program, Sun JVM 1.4.2 is worse than IBM JVM 1.4.2 for the *ack* program but it is better for the *fibo* program.

With Romizing technique, the runtime systems class is embedded in its core, KVM needs only 0 millisecond (rounded up value) for the *HelloWorld* program compared with a few dozen milliseconds in the cases of other JVMs.

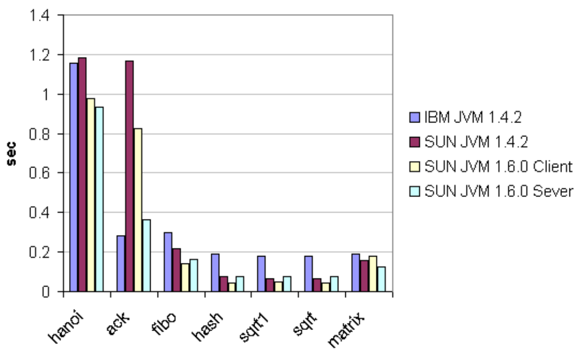


Fig. 9. Runtime comparison, the lower the better.

Categories	Name	Description
Simple Arithmetic	SquareRoot1 Nestedloop	Integer Square root of 27 with Newton method 6 nested for loop with index value is 10
Large computation	MatrixArr1 Matrix	Fill the int array of 5000 elements with value 1 Multiply two dimension matrices of integers, each has a size of 30
Multiple stacks call	SquareRoot HelloWorld Random Fibo Quicksort Towers	integer Square root of 27 with recursive calls implementation Print string Hello World to standard output Generate random 100 double numbers and print out Fibonacci for 35, with recursive calls implementation Sort an array which has 100 numbers by Quick sort algorithm Hanoi Tower for 15 disks
Object oriented	HashMap1 Shellsort HashMap2 BankAccout FFT OO	Create HashMap, add 10 elements to HashMap Shellsort with object oriented implementation, 10 random elements Create a HashMap with 1000 element, copy to other HashMap Bank Account list with 1000 members Fast Fourier Transform with OO implemented

TABLE II
FHOJ WORKLOAD - TESTSET

2) Performance of the JIT versus the Interpreter: JVMs which use the JIT architecture is likely to have longer start-up times because the JIT compiler has to compile Java bytecodes into native machine language before executing.

Nevertheless, ADC and JIT execution engine prove to be far more better than the Interpreter architecture when applications have to deal with large computation or multiple stack called, see Figure 10. For complex programs, KVM spends from 10 up to 50 times more than other JVMs.

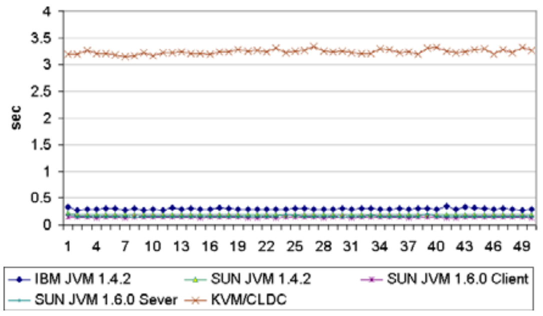


Fig. 10. Fibonacci up to the 35th number. The lower the better. Show the advance of the JIT compare with the Interpreter execution engine in large computation applications

3) Standard runtime deviation: The standard deviations of JVMs for Hanoi tower are calculated using the formula 1 and 2, see table III. The second column shows the values in FHOJ, the third column shows the values when we measure time by using the traditional method. The third column shows how better the values in FHOJ framework in percent. The standard deviation of the benchmark using the FHOJ framework for the Hanoi Tower program is about 7% lower than the values which are archieved from the tranditional way to do time measurement in other benchmarks.

Note that the standard deviation achieved using the FHOJ framework will be increased or decreased depending on the running services in this computer. The higher number of the running services in the system is, the better values of the FHOJ framework will be. Table IV shows the comparison when all

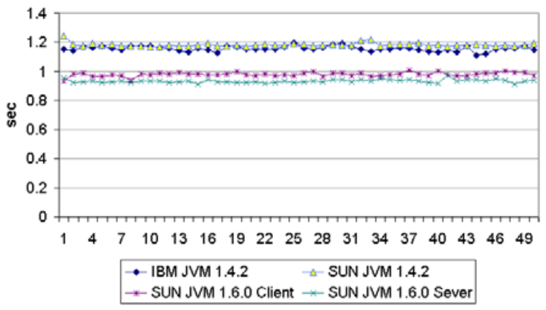


Fig. 11. Runtime comparison, Hanoi 27 towers. The lower the better.

Vendor	FHOJ	Td. method	Pc(%)
IBM 1.4.2	0.0016970	0.0019234	13.34
SUN 1.4.2	0.0012467	0.0013345	7.05
SUN 1.6.0 Client	0.0013202	0.0014013	6.14
SUN 1.6.0 Sever	0.0011501	0.0012223	6.28

TABLE III
STANDARD RUNTIME DEVIATION COMPARISION OF JVMS FOR THE HANOI TOWER PROGRAM.

services in this system were set in *enable*. Both results of Table III and Table IV are produced by the same hardware configuration.

Vendor	FHOJ	Td. method	Pc(%)
IBM 1.4.2	0.0031720	0.0035346	10.26
SUN 1.4.2	0.0022331	0.0026633	16.15
SUN 1.6.0 Client	0.0021272	0.0024873	14.48
SUN 1.6.0 Sever	0.0028501	0.0034582	17.58

TABLE IV
STANDARD RUNTIME DEVIATION COMPARISION OF JVMS FOR THE HANOI TOWER PROGRAM WHEN MORE SEVICES IN THE SYSTEM WERE ACTIVATED.

V. CONCLUSION

FHOJ Benchmark has been designed and implemented, the advance features of FHJO, that directly deal with behaviors of multi tasks operating systems. The variances of the runtime measurement in FHOJ are proved, in some cases, to be about 7% more reliable than other benchmarks.

By introducing an add-on module, the BenchmarkCrt, FHOJ has separated processes of the workload and output results. The changes in parameters in benchmark such as: input values to workload, execution times, is done by changing options in the BenchmarkCrt, and will not effect to the precise values of the output.

By deploying both a high resolution timer [2] and a fine grained accounting system [12], we can improve the precision of the FHOJ framework's results. However, in this way a significant work must be done since we have to re-write the kernel schedule management of the OS.

Developing further automation steps in the framework such as: reports collection and reports generation is also necessary to do. Collection data from a large number of running times should not done manually to produce the reports.

The TestSet or workload for the FHOJ currently is not inclusive, that means the applications used in FHOJ do not really stand for daily applications. It is also a common problem of other benchmarks. Therefore, further analysis of the workload should be done.

VI. Acknowledgment

The author would like to thank Prof. Dr. Jan-Wilhelm Fischer of the University Of Applied Sciences Offenburg for his help and support in making this work possible.

References

[1] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.
[2] H. resolution timers, "http://high-res-timers.sourceforge.net/." [Online]. Available: http://high-res-timers.sourceforge.net/
[3] J. P. Charles Daly, Jane Horgan and J. Waldron, "Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite," *In Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pp. 106–115, 2001.
[4] M. D. W. D. S. H. J. M. Bull, L. A. Smith and R. A. Davey, "A Methodology for Benchmarking Java Grande Applications," *In Proceedings of the ACM 1999 conference on Java Grande*, pp. 81–88, 1999.
[5] P. S. Corporation, "CaffeineMark," *http://www.benchmarkhq.ru/cm30/*, 2002.
[6] S. Org, "Spec JVM 98," *http://www.spec.org/osg/jvm98*, 1998.
[7] X. Zhang and M. Seltzer, "HBench: Java: An Application-Specific Benchmarking Framework for Java Virtual Machine," *In Proceedings of the ACM 2000 conference on Java Grande*, 2000.
[8] R. W. Jack Dongarra and P. McMahan, "LINPACK Benchmarks," *http://www.netlib.org/benchmark/linpackjava/*.
[9] N. V. G. Chen, M. Kandemir and M. J. Irwin, "PennBench: A Benchmark Suite for Embedded Java," *IEEE International Workshop on Workload Characterization*, pp. 71–80, 2002.
[10] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 169–190.
[11] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake up and smell the coffee: evaluation methodology for the 21st century," *Commun. ACM*, vol. 51, no. 8, pp. 83–89, 2008.
[12] M. S. Shuichi Oikawa and T. Nakajima, "Accounting system: a fine-grained CPU resource protection mechanism for embedded system," *In Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.
[13] G. M. I. Bate, G.Bernat and P. Puschner, "Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework," *In Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, 2000.
[14] Jack Shirazi and Kirk Pepperdine, "Eye on performance: Micro performance benchmarking," *IBM journal*, 2003.
[15] Jack Shizazi and Kirk Pepperdine, "Eye on performance: When good benchmark go bad," *IBM journal*, 2005.
[16] A. G. Lieve Eeckhout and K. D. Bosschere, "How Java Programs Interact with Virtual Machines at the Microarchitectural Level," *In Proceedings of the OOPSLA 03 conference*, 2003.
[17] R. Pozo and B. Miller, "SciMark 2.0 Benchmark," *http://math.nist.gov/scimark2*, 2004.
[18] M. G. Yefim Shuf, Maurico J. Serrano and J. P. Singh, "Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations," *ACM Sigmetrics*, 2001.
[19] S. M. Y. T. T. Y. Morgan Hirosuke Miki, Mamoru Sakamoto and I. Shirakawa, "Evaluation of Processor Code Efficiency for Embedded Systems," *In Proceedings of the 15th international conference on Supercomputing*, 2001.
[20] V. Q. La, "Design Virtual Machine for Java Processing for a Small Embedded Microprocessor Core," Master's thesis, University Of Applied Sciences Offenburg, 2007.
[21] V.-Q. La, "A study on Java Virtual Machine for Real-time embedded systems," *IEEE International Conference on Computer Science and Software Engineering (CSSE 2008)*, Accepted 2008.