Power-Performance Analysis of JVM Implementations

Hitoshi Oi
Department of Computer Engineering
The University of Aizu,
Aizu Wakamatsu, JAPAN
Email: oi@oslab.biz

Abstract—**Java Virtual Machines (JVMs) work between the Java application programs and the operating systems (and their underlying hardware platforms) to provide the 'write-once run-anywhere' property of the Java language. However, this property also implies that the runtime efficiency, in terms of both performance and power-consumption, can be affected by the implementations of JVM.**

In this paper, we present a case study of performance-power analysis of JVM implementations. We run SPECjvm2008 with OpenJDK and IBM I9 on an Atom-based netbook with Ubuntu operating system. Our observations are as follows: (1) the relative performance of OpenJDK ranges from 44 to 289% of J9, (2) the dynamic power consumption ranges from 2.8 to 7.2 Watts among benchmark programs. However, the power consumptions of two JVMs for the same workload are relatively similar. Therefore, the power-performance efficiency is mostly affected by the relative performance. (3) the effectiveness of multi-threading varies among benchmark programs as well as among JVMs. In general, running benchmark with a single-thread loses more in performance than in power consumption. Exceptions are compress, fft.small (OpenJDK only) and lu.small. (4) for most benchmark programs (except scimark), the power consumptions seem to be correlated to linear and square root of L2 reference and L2 miss rates, respectively.

Keywords SPECjvm2008, Performance Evaluation, Power Consumption

I. INTRODUCTION

Java is one of the most popular and standard programming languages these days. It runs on various platforms, such as smart phones, to the data center servers handling a huge number of requests each second. One of the reasons of Java's popularity can be attributed to the Java Virtual Machine (JVM), which is a virtual instruction set architecture [1] of Java. Unlike other high-level programming languages, that are compiled into the machine languages of the underlying processors, Java applications are complied into JVM's instructions, called Java bytecodes.

JVMs need to perform the the operations specified by the bytecodes of the applications. The details of JVM implementation are left flexible: some are optimized for the execution time and some others are designed for small memory footprints. However, due to this flexibility of JVM implementation, the runtime behavior of Java applications can be affected by various factors, such as:

(1) When a Java application is started, JVM is invoked and its own data structures are initialized. (2) JVM plays the role of memory management, especially the garbage collection. (3) modern JVMs utilize dynamic compilation techniques: Consequently, the performance and power-efficiency of Java applications on the same hardware platform with the same operating system can be quite different depending on the JVM.

In this paper, we present a case study of analyzing JVM implementations in terms of performance and power consumption. As the workload, we use SPECjvm2008, which is a benchmark suite from SPEC for evaluating client-side JVMs [2], on two popular open-source JVMs, OpenJDK [3] and IBM J9 [4]. In addition to the executions in the base configuration, we present the effects of multi-threading and slower clock speed and the correlation of cache reference parameters to the power consumption.

This paper is organized as follows. In the next section, the workload of SPECjvm2008 is described. The measurement results and their analysis, including the comparisons between OpenJDK and J9 and the effects of multi-threading and clock speed, are presented in Section III. Related work are introduced in Section IV and the paper concludes in SectionV.

Disclosure

SPECjvm2008 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The use of SPECjvm2008 in this paper falls into the "Research and Academic Usage of SPECjvm2008" in [5]. The results of running SPECjvm2008 presented in this paper are not audited by SPEC and must not be compared to any officially published results from SPEC.

II. SPECJVM2008

In this section, a brief description of SPECjvm2008 workload is presented. SPECjvm2008 consists of 38 benchmark programs that are classified into the eleven categories listed in Table I. *compiler* consists of two benchmarks, compiler.compiler and compiler.sunflow. The former compiles `javac` itself and the latter compiles another

978-1-4577-0989-0/11/$26.00 ©2011 IEEE

benchmark program, sunflow, in SPECjvm2008. *compress* is a compression workload based on Lempel-Ziv method (LZW). This program is ported from SPEC CPU95 [6], but the input is a real data rather than the synthesized one in SPEC CPU95. *crypto* has three sub-benchmark programs of encryption, decryption and sign-verification using different protocols: crypto.aes (AES and DES protocols), crypto.rsa (RSA) and crypto.signverify (MD5withRSA, SHA1withRSA, SHA1withDSA and SHA256withRSA). *derby* is a database benchmark in Java and it is to replace the `db` benchmark in SPEC JVM98. It is designed to represent a more realistic application than db and to stress the BigDecimal library. *mpegaudio* is an mp3 decoding benchmark and corresponds to the benchmark with the same name in JVM98. The mp3 library of mpegaudio in JVM98 has been replaced with JLayer [7]. It evaluates the floating-point operations of the JVM. SciMark is a computational benchmark suite in Java developed by NIST [8]. It consists of five sub benchmark programs (fft, lu, monte_carlo, sor and sparse). In SPECjvm2008, they are executed with a large data set (32MB) for testing the memory hierarchy and a small (512KB) data set for testing the JVM itself. *serial* operates in a producer-consumer scenario, where the producer serializes primitives and objects from JBoss benchmark and sends them over the socket to the consumer. These data are deserialized at the consumer. In *startup*, a new JVM is started for each benchmark in SPECjvm2008 and it runs one iteration of the benchmark. The time from starting up the JVM to the end of the benchmark iteration is measured. *sunflow* is a multi-threaded rendering benchmark program. It starts with half the number of threads as the number of logical CPUs, and each of these threads spawns four threads inside the program. *xml* is made of two sub-benchmark programs: *xml.transform* and *xml.validation*. The former evaluates the implementation of *java.xml.transform* of the JVM under test, while the latter uses *java.xml.validation* to compare the XML files and corresponding XML specifications in .xsd files. SPEC defines two categories to run SPECjvm2008: Base and Peak. While no optimization is allowed in the Base category, any optimizations are allowed in the Peak category. In this paper, we run the benchmark in the base category, i. e. no optimization is used, except limiting the number of threads to one in Section III-C. Except startup, each benchmark is executed for six minutes: first two minutes are for warming-up the JVM and the rest are for the actual measurement.

III. Power-Performance Analysis

In this section, we first describe the platform used for the measurement, and then present the results of measurement and analysis.

Category	Description & Sub-Benchmarks
compiler	Compilation of .java files. compiler.compiler, compiler.sunflow
compress	Compression by LZW method.
crypto	Encryption and decryption. crypto.aes, crypto.rsa, crypto.signverify
derby	Database focused on BigDecimal.
mpegaudio	Mp3 decoding.
scimark.large scimark.small	Floating point benchmark with 32MB and 512KB datasets. scimark.fft.large, scimark.fft.small scimark.lu.large, scimark.lu.small scimark.monte_carlo, scimark.sor.large scimark.sor.small, scimark.sparse.large scimark.sparse.small
serial	Primitive and object (de)serializations.
startup	JVM launch time for each benchmark. startup.compiler.compiler, startup.compiler.sunflow startup.compress, startup.crypto.aes startup.crypto.rsa, startup.crypto.signverify startup.helloworld, startup.mpegaudio startup.scimark.fft, startup.scimark.lu startup.scimark.monte_carlo, startup.scimark.sor startup.scimark.sparse, startup.serial startup.sunflow, startup.xml.transform startup.xml.validation
sunflow	Graphics visualization (rendering).
xml	XML transform and validation. xml.transform, xml.validation

Table I
SPECjvm2008 Workload Categories

A. Experimental Environment

Table II shows the specifications of the hardware and software of the measurement platform. We use a netbook with an Intel Atom N270 which is a single core CPU with Hyperthreading [9], running Ubuntu operating system. For the measurement of power consumption, we use a Watts up ? Pro 99333 power meter [10]. The monitor screen is darkened by the screen saver as SPECjvm2008 does not use any graphics.

We use two JVM implementations, OpenJDK [3] and IBM J9 [4]. OpenJDK is an open-source implementation of Java language begun by Sun Microsystems in 2006. We use IcedTea6 1.9.5 version which came with the Ubuntu operating system. J9 is an implementation of Java by IBM available for various platforms, such as Power7 or System Z.

B. Measurement Results of Base Configuration

Table III shows the performance and power consumption of OpenJDK and J9. For the performance metrics, we use the number of iterations that each benchmark can execute per minute (I). When the CPU is running with Hyperthreading, the measurement platform (Dell Inspiron Mini 9) consumes 6.9W of power in the idle state. W in Table III indicates the increase of the power consumption from the idle state. I and

Component	Specification
Platform	Dell Inspiron Mini 9
CPU	Atom N270 (1.6GHz, TDP 2.5W)
Cache Sizes	32KB (L1I), 24KB (L1D), 512KB (L2)
Memory	1GB
OS	Ubuntu 10.04.2 LTS (kernel 2.6.32-28)
OpenJDK	Java 1.6.0_20 IcedTea6 1.9.5
J9	Java 1.6.0 Build 2.4, SR9-FP1
Power Meter	Watts up ? Pro 99333

Table II
MEASUREMENT ENVIRONMENT

W in Ratios represent these metrics of OpenJDK divided by those of J9. Eff is (I/W) of OpenJDK divided by that of J9.

First, we see that the relative performance between JVMs varies among benchmark programs: OpenJDK is 289% of J9 for startup.helloworld, but it is only 44% for scimark.monte_carlo. Second, the power consumption varies largely among benchmark programs, but not between two JVMs. As a result, performance-power ratios are close to the relative performance. Among benchmark categories, scimark.* have high power consumptions. The exception is monte_carlo. It is a π computation program using randomly generated points in a square. From our past experience [11], it is known to have shorter loop-bodies in bytecode (on average, 54 bytecodes while other scimark.* are 300 or more bytecodes long) that leads to frequent method invocations and fewer chances of local variable reuses. These two characteristics make monte_carlo more similar to other benchmarks than to scimark.*. In scimark.*, there are two data sets; large and small. Intuitively, *.large benchmarks should consume more power than *.small as the former place more pressure on the memory hierarchy. sparse is the exception to this intuition: it consumes 24 and 10% more power for the smaller data sets on OpenJDK and J9, respectively.

C. Effect of Multi-Threading

By default, SPECjvm2008 workloads spawn the number of threads equal to the number of logic CPUs of the system. Since we use an Atom N270 which is a single core with Hyperthreading, two threads are spawned and execute workloads. There are two exceptions for this default number of benchmark threads: `startup.*` and `sunflow` benchmarks. The former is single-threaded and the latter spawns half the number of logical CPUs (and each of them splits into four threads inside the benchmark). In this section, to investigate the effectiveness of multi-threading, we limit the number of benchmark thread to one with the `-bt 1` option.

Table IV shows the measurement results of single-thread executions. The effectiveness of multi-threading in performance varies significantly among benchmark programs as well as between JVMs. The numbers in parentheses below

	OpenJDK		J9		Ratios		
Benchmark	I	W	I	W	I	W	Eff
compiler							
compiler	17.2	3.8	21.4	4.1	0.81	0.93	0.87
sunflow	6.3	3.8	8.5	4.0	0.73	0.93	0.79
compress	10.1	4.2	10.6	4.3	0.95	0.98	0.97
crypto							
crypto.aes	2.9	3.1	6.3	3.2	0.45	0.99	0.46
crypto.rsa	6.1	3.0	11.7	3.2	0.52	0.95	0.55
crypto.signverify	10.3	3.2	15.4	3.3	0.67	0.97	0.69
derby	7.8	3.5	5.8	3.9	1.35	0.89	1.52
mpegaudio	4.8	3.2	8.1	3.4	0.60	0.94	0.64
scimark							
fft.large	4.4	6.5	4.5	6.5	0.98	1.01	0.97
fft.small	8.7	6.5	9.8	6.4	0.89	1.02	0.87
lu.large	1.2	7.2	1.3	7.2	0.90	1.00	0.91
lu.small	10.0	6.2	11.7	6.1	0.86	1.01	0.85
monte_carlo	4.6	3.2	10.4	3.1	0.44	1.02	0.43
sor.large	3.0	5.0	4.5	5.6	0.69	0.90	0.76
sor.small	13.5	5.0	20.4	5.5	0.66	0.92	0.72
sparse.large	2.2	5.0	2.4	5.1	0.90	0.98	0.92
sparse.small	10.2	6.2	8.3	5.6	1.22	1.10	1.11
serial	4.6	3.4	5.6	3.7	0.82	0.92	0.89
sunflow	4.0	3.4	4.1	3.5	0.97	0.96	1.01
xml							
xml.transform	10.7	3.6	9.0	4.0	1.19	0.89	1.33
xml.validation	15.6	3.5	23.1	3.8	0.67	0.91	0.74
startup							
compiler.compiler	3.2	3.5	1.3	4.0	2.53	0.88	2.88
compiler.sunflow	2.7	3.5	1.3	4.0	2.00	0.88	2.29
compress	7.0	3.2	5.3	3.6	1.33	0.90	1.48
crypto.aes	2.0	2.9	2.3	3.5	0.84	0.83	1.01
crypto.rsa	4.2	2.9	3.6	3.6	1.17	0.80	1.46
crypto.signverify	6.5	2.9	3.2	3.4	2.08	0.87	2.39
helloworld	103.2	3.8	35.8	3.8	2.89	1.00	2.89
mpegaudio	2.9	2.8	2.6	3.7	1.15	0.76	1.52
fft	7.2	4.0	6.0	4.2	1.20	0.96	1.24
lu	6.3	3.5	5.5	3.7	1.14	0.92	1.24
monte_carlo	2.8	2.9	3.7	3.1	0.75	0.95	0.79
sor	7.0	3.4	7.4	3.5	0.94	0.96	0.98
sparse	6.3	4.3	4.6	4.1	1.37	1.05	1.30
serial	3.0	3.1	2.0	4.0	1.49	0.79	1.88
sunflow	3.4	3.3	1.5	3.6	2.24	0.92	2.42
xml.transform	0.4	2.8	0.3	3.3	1.51	0.84	1.80
xml.validation	6.4	3.3	3.2	3.9	1.97	0.84	2.36

Table III
BASE CONFIGURATION MEASUREMENT RESULTS. I IS THE NUMBER OF ITERATIONS EACH BENCHMARK RUNS PER MINUTE. W IS THE DYNAMIC POWER (INCREASE FROM THE IDLE STATE) IN WATTS. I AND W IN RATIOS REPRESENT METRICS OF OPENJDK DIVIDED BY J9, RESPECTIVELY. EFF IN RATIOS REPRESENTS THE RATIO OF I/W BETWEEN OPENJDK AND J9.

I in Table IV represent the relative performance of single-thread executions. In terms of performance, the smaller the this number, the more effective to run the benchmark with two threads. The most typical example is scimark.sor.large on J9, whose performance is almost halved with a single thread. For the difference between JVMs, we can see two extreme cases in xml.transform and scimark.fft.small: in the former benchmark, multi-threading is more effective in OpenJDK (71% vs 88%) but the opposite is true for the latter benchmark (82% vs 74%).

In terms of performance-power consumption balance, the

Benchmark	OpenJDK I	OpenJDK W	OpenJDK Eff	J9 I	J9 W	J9 Eff
compiler						
compiler	13.3 (0.77)	3.3 (0.87)	0.89	18.0 (0.84)	3.5 (0.86)	0.98
sunflow	5.1 (0.82)	3.3 (0.88)	0.92	7.0 (0.82)	3.5 (0.88)	0.93
compress	8.0 (0.79)	3.1 (0.74)	1.07	8.3 (0.79)	3.1 (0.72)	1.09
crypto						
aes	2.0 (0.70)	2.8 (0.91)	0.78	4.1 (0.65)	2.8 (0.88)	0.74
rsa	4.7 (0.77)	2.9 (0.94)	0.82	8.3 (0.71)	2.8 (0.88)	0.81
signverify	7.8 (0.76)	2.9 (0.92)	0.82	11.4 (0.74)	2.9 (0.88)	0.84
derby	5.2 (0.66)	3.0 (0.88)	0.76	3.6 (0.62)	3.4 (0.87)	0.71
mpegaudio	3.1 (0.65)	2.8 (0.87)	0.74	5.1 (0.62)	2.9 (0.86)	0.73
scimark						
fft.large	3.1 (0.69)	5.1 (0.78)	0.89	3.0 (0.66)	5.1 (0.79)	0.83
fft.small	7.1 (0.82)	4.5 (0.69)	1.18	7.3 (0.74)	4.4 (0.70)	1.06
lu.large	0.8 (0.65)	5.6 (0.78)	0.84	0.8 (0.60)	5.4 (0.75)	0.81
lu.small	6.8 (0.68)	3.6 (0.58)	1.17	7.7 (0.66)	3.4 (0.55)	1.18
monte_carlo	3.0 (0.66)	2.9 (0.91)	0.72	6.6 (0.64)	2.8 (0.90)	0.71
sor.large	1.7 (0.55)	3.8 (0.75)	0.74	2.4 (0.54)	4.0 (0.71)	0.76
sor.small	7.5 (0.56)	3.3 (0.66)	0.85	11.7 (0.57)	3.4 (0.61)	0.93
sparse.large	1.4 (0.62)	3.9 (0.79)	0.79	1.3 (0.54)	3.8 (0.76)	0.71
sparse.small	6.9 (0.68)	4.9 (0.79)	0.86	6.0 (0.72)	4.5 (0.81)	0.89
serial	3.4 (0.75)	3.1 (0.91)	0.83	3.9 (0.70)	3.3 (0.89)	0.78
sunflow	4.0 (1.00)	3.5 (1.03)	0.97	4.1 (1.00)	3.6 (1.02)	0.98
xml						
transform	7.6 (0.71)	3.1 (0.88)	0.80	7.9 (0.88)	3.6 (0.91)	0.97
validation	10.5 (0.67)	3.1 (0.90)	0.75	17.0 (0.74)	3.3 (0.87)	0.85

Table IV
MEASUREMENT RESULTS OF SINGLE-THREAD EXECUTIONS. NUMBERS IN PARENTHESES BELOW I AND W ARE THE RATIOS OF I AND W AGAINST THOSE OF DUAL THREAD EXECUTIONS (TABLE III). EFF IS I/W OF SINGLE-THREAD DIVIDED BY THAT OF DUAL THREAD.

relative performances of single-thread executions have a different meaning. If the multi-threading is not very effective in terms of performance, we may choose to run the program with a single thread to expect power reduction. Eff columns in Table IV present the ratios between relative performance and power consumption (ratios of I/W in dual and single thread executions). In general, we lose more in performance than in power-consumption. Exceptions are compress, fft.small (OpenJDK only) and lu.small. Like dual-thread executions, the difference of power consumption between JVMs is small and it is the relative performance which determines the effectiveness of choosing single-thread execution. The most typical case is fft.small; its relative performance of single-thread execution on OpenJDK is 84% which is higher than the relative power consumption (68%). The relative performance of the same benchmark on J9 is only 66% and we see that the single-thread execution is not a feasible option for J9 in terms of performance-power balance.

D. *Effect of Clock Speed*

Atom N270 is equipped with the Enhanced Intel Speed-Step Technology [12], which enables the system to dynamically adjust the clock speed and the voltage. The measurements in the previous sections have all been executed at Atom N270's default maximum clock frequency (1.6GHz). In this section, we limit the clock frequency to 800MHz using `cpufreq-set` [13] and analyze the effect of lower clock frequency.

Tables V and VI show the performance and power consumption at the clock frequency of 800MHz. We use the same notations as Table IV for I, W and Eff. With few exceptions, the relative performance is around 50% of the 1.6GHz execution. First type of exceptions include scimark.fft and scimark.lu whose performance are significantly higher than 50% (68 to 75%). As we will see in the next subsection, for scimark.fft, the high L2 miss rates could be the reason; slowing down the clock frequency makes the relative speed of memory access faster (i.e. lower miss penalty). However, this reason is not applicable (at least, naively) to scimark.lu, since there are seven benchmarks that have higher L2 miss rates than scimark.lu on both JVMs. Another type of exception is xml.validation on J9, whose performance is reduced to 37% of 1.6GHz. Unlike scimark.fft and scimark.lu, this performance degradation only happened to J9 as the relative performance of OpenJDK is 52%.

The relative power consumption ranges from 52% (scimark.sparse.large on J9) to 72% (scimark.fft.large on both JVMs). As mentioned above, scimark.fft and scimark.lu are not slowed down as other benchmarks with 800MHz clock and they have relatively high cache reference and miss rates. Therefore, it is considered that these workloads have high utilizations of both functional units and memory hierarchies. Efficiencies (Eff) are around 90% or higher. Exceptions are compiler.* and xml.* (and some of startup.*) on J9, which should be the results of their lower performances than 50% at 800Mhz clock frequency.

E. *Microoperation and Cache Reference*

Using Oprofile (version 0.9.6) [14], we have measured the numbers of retired micro-operation (U), L2 cache reference (R) and L2 cache miss (M) against 1, 10 and 100 clock cycles as shown in Table VII. First, we could not have found

Benchmark	OpenJDK			J9		
	I	W	Eff	I	W	Eff
compiler						
compiler	9.3 (0.54)	2.2 (0.57)	0.94	9.2 (0.43)	2.4 (0.58)	0.74
sunflow	3.3 (0.53)	2.1 (0.57)	0.93	3.6 (0.42)	2.3 (0.57)	0.73
compress	5.4 (0.54)	2.4 (0.57)	0.94	5.8 (0.55)	2.6 (0.59)	0.93
crypto						
aes	1.4 (0.50)	1.7 (0.55)	0.92	3.2 (0.51)	1.8 (0.56)	0.90
rsa	3.0 (0.50)	1.6 (0.54)	0.93	5.7 (0.49)	1.7 (0.53)	0.93
signverify	5.2 (0.50)	1.7 (0.54)	0.93	7.6 (0.50)	1.7 (0.53)	0.93
derby	4.0 (0.51)	2.0 (0.57)	0.90	2.9 (0.50)	2.2 (0.58)	0.86
mpegaudio	2.4 (0.50)	1.7 (0.55)	0.91	4.0 (0.49)	1.9 (0.55)	0.88
scimark						
fft.large	3.3 (0.75)	4.7 (0.72)	1.04	3.4 (0.75)	4.7 (0.72)	1.04
fft.small	6.2 (0.71)	4.4 (0.68)	1.06	6.7 (0.68)	4.3 (0.67)	1.02
lu.large	0.7 (0.55)	3.9 (0.53)	1.03	0.8 (0.63)	4.2 (0.58)	1.10
lu.small	5.4 (0.54)	3.4 (0.54)	0.98	6.5 (0.56)	3.5 (0.58)	0.97
monte_carlo	2.3 (0.50)	1.7 (0.53)	0.94	5.2 (0.50)	1.7 (0.55)	0.91
sor.large	1.5 (0.51)	2.6 (0.52)	0.98	2.3 (0.51)	2.9 (0.52)	0.98
sor.small	6.8 (0.50)	2.6 (0.53)	0.96	10.3 (0.50)	3.0 (0.55)	0.92
sparse.large	1.2 (0.53)	2.7 (0.54)	0.98	1.0 (0.41)	2.6 (0.52)	0.79
sparse.small	5.2 (0.51)	3.2 (0.51)	0.99	4.2 (0.50)	3.0 (0.53)	0.96
serial	2.3 (0.50)	1.9 (0.57)	0.89	2.4 (0.43)	2.2 (0.59)	0.72
sunflow	2.0 (0.50)	2.0 (0.58)	0.87	2.1 (0.50)	2.1 (0.58)	0.86
xml						
xml.transform	5.4 (0.51)	2.0 (0.57)	0.89	4.0 (0.45)	2.3 (0.57)	0.78
xml.validation	8.1 (0.52)	2.0 (0.57)	0.91	8.5 (0.37)	2.3 (0.60)	0.62

Table V
PERFORMANCE AND POWER CONSUMPTION AT 800 MHZ CLOCK

Benchmark	OpenJDK			J9		
	I	W	Eff	I	W	Eff
startup						
c.compiler	1.7 (0.53)	2.1 (0.60)	0.88	0.6 (0.50)	2.3 (0.58)	0.86
c.sunflow	1.4 (0.53)	2.1 (0.60)	0.88	0.6 (0.48)	2.3 (0.57)	0.84
compress	3.6 (0.51)	1.9 (0.59)	0.87	2.8 (0.54)	2.1 (0.57)	0.94
crypto.aes	1.0 (0.50)	1.7 (0.59)	0.86	1.2 (0.50)	2.0 (0.57)	0.88
crypto.rsa	2.1 (0.51)	1.6 (0.57)	0.89	1.7 (0.48)	2.0 (0.57)	0.84
crypto.signverify	3.3 (0.51)	1.7 (0.58)	0.88	1.8 (0.56)	1.9 (0.56)	0.99
helloworld	54.9 (0.53)	2.1 (0.55)	0.96	21.2 (0.59)	2.2 (0.58)	1.03
mpegaudio	1.5 (0.50)	1.6 (0.58)	0.86	1.2 (0.45)	2.1 (0.56)	0.81
s.fft	4.1 (0.56)	2.6 (0.64)	0.88	3.5 (0.59)	2.7 (0.64)	0.92
s.lu	3.2 (0.51)	2.0 (0.58)	0.87	2.9 (0.52)	2.2 (0.59)	0.89
s.monte_carlo	1.4 (0.51)	1.6 (0.56)	0.91	1.9 (0.50)	1.7 (0.55)	0.91
s.sor	3.5 (0.50)	2.0 (0.58)	0.87	3.9 (0.52)	2.1 (0.59)	0.89
s.sparse	3.2 (0.51)	2.5 (0.58)	0.87	2.4 (0.52)	2.4 (0.58)	0.89
serial	1.5 (0.51)	1.8 (0.58)	0.88	1.0 (0.49)	2.2 (0.57)	0.86
sunflow	1.7 (0.51)	1.9 (0.57)	0.89	0.6 (0.41)	2.0 (0.56)	0.74
xml.transform	0.2 (0.51)	1.7 (0.61)	0.84	0.1 (0.48)	2.0 (0.59)	0.81
xml.validation	3.4 (0.53)	2.0 (0.61)	0.87	1.5 (0.46)	2.2 (0.57)	0.81

Table VI
PERFORMANCE AND POWER CONSUMPTION AT 800 MHZ CLOCK (2)

potential correlations between U and relative performance or power consumption. In Java applications, a higher U does not always mean a higher (application) performance. This is because, for example, a JVM may have to execute more native instructions than another JVM for the same Java bytecode. In other words, a higher U may be the result of inefficient bytecode interpretation/compilation. For example, Us for crypto.* in J9 are lower than those in OpenJDK. However, the performances of J9 for these benchmarks are better than OpenJDK. Moreover, despite discrepancies in Us, two JVM consume almost the same power (Table III).

Figure 1 and 2 plot the pairs of the power consumption and L2 cache reference and miss rates (against clock cycles), respectively. In these graphs, scimark.* (except scimark.monte_carlo) and other benchmark programs occupy different regions (as a result of scimark.*'s higher power consumption). Intuitively, the power consumptions of non-scimark benchmarks seem to be correlated to linear and square root of L2 reference rate and L2 miss rate. Using the fit command of gnuplot, functions for power consumption by L2 reference rate (r) and L2 miss rate (m) are obtained and plotted in Figure 1 and 2, respectively. Generally, data points look to fit well to the functions with few exceptions such as startup.mpegaudio at $(2.4 \times 10^{-3}, 2.83)$ for OpenJDK or startup.scimark.sparse at $(4.930 \times 10^{-3}, 4.09)$ for J9 in Figure 1.

IV. RELATED WORK

In [15], authors evaluated the performance and power of Java applications on IA-32 microprocessors with fabrication technologies ranging from 120nm to 32nm. For the single and multi-thread benchmarks, they used SPEC JVM98 and and SPECjbb2005 (and other server applications), respec-

Benchmark	OpenJDK			J9		
	U	R	M	U	R	M
compiler						
compiler	0.26	0.14	0.12	0.30	0.12	0.15
sunflow	0.28	0.12	0.09	0.32	0.11	0.12
compress	0.38	0.14	0.23	0.36	0.15	0.25
crypto						
aes	0.69	0.01	0.01	0.63	0.01	0.02
rsa	0.73	0.00	0.00	0.62	0.01	0.01
signverify	0.75	0.01	0.01	0.68	0.01	0.02
derby	0.32	0.11	0.07	0.30	0.12	0.11
mpegaudio	0.56	0.03	0.01	0.54	0.04	0.02
scimark						
fft.large	0.15	0.08	0.48	0.11	0.09	0.49
fft.small	0.25	0.10	0.42	0.18	0.11	0.41
lu.large	0.49	0.09	0.08	0.32	0.09	0.08
lu.small	0.55	0.07	0.09	0.32	0.08	0.11
monte_carlo	0.58	0.00	0.00	0.59	0.00	0.00
sor.large	0.50	0.06	0.02	0.29	0.09	0.03
sor.small	0.56	0.03	0.02	0.31	0.04	0.03
sparse.large	0.39	0.07	0.16	0.31	0.07	0.18
sparse.small	0.63	0.07	0.05	0.67	0.06	0.05
serial	0.37	0.07	0.03	0.38	0.09	0.07
sunflow	0.42	0.04	0.02	0.42	0.04	0.04
xml						
transform	0.40	0.09	0.05	0.35	0.10	0.08
validation	0.29	0.15	0.06	0.33	0.13	0.08
startup						
compiler.compiler	0.22	0.09	0.10	0.33	0.10	0.11
compiler.sunflow	0.23	0.08	0.09	0.33	0.10	0.11
compress	0.30	0.09	0.06	0.32	0.09	0.08
crypto.aes	0.51	0.01	0.01	0.43	0.05	0.05
crypto.rsa	0.54	0.01	0.01	0.43	0.06	0.06
crypto.signverify	0.53	0.01	0.01	0.42	0.09	0.04
helloworld	0.25	0.06	0.08	0.32	0.07	0.09
mpegaudio	0.38	0.02	0.01	0.43	0.09	0.06
scimark.fft	0.21	0.08	0.22	0.20	0.08	0.20
scimark.lu	0.37	0.05	0.03	0.29	0.06	0.06
scimark.monte_carlo	0.39	0.01	0.01	0.35	0.02	0.02
scimark.sor	0.32	0.02	0.02	0.27	0.04	0.05
scimark.sparse	0.42	0.05	0.03	0.43	0.05	0.04
serial	0.26	0.06	0.04	0.35	0.10	0.11
sunflow	0.39	0.05	0.03	0.38	0.07	0.05
xml.transform	0.20	0.07	0.02	0.26	0.10	0.05
xml.validation	0.22	0.10	0.08	0.33	0.10	0.10

Table VII
RATES OF RETIRED MICROOPERATION (U), L2 CACHE REFERENCE (R) AND L2 CACHE MISS (M) PER 1, 10 AND 100 CLOCK CYCLES, RESPECTIVELY

.

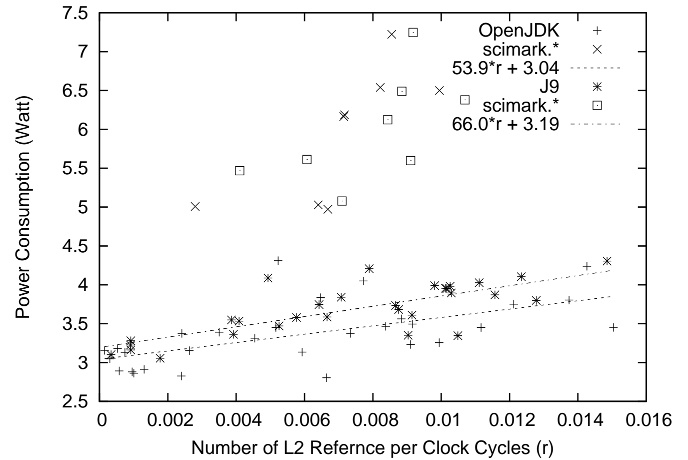

Figure 1. L2 Cache Reference and Dynamic Power Consumption

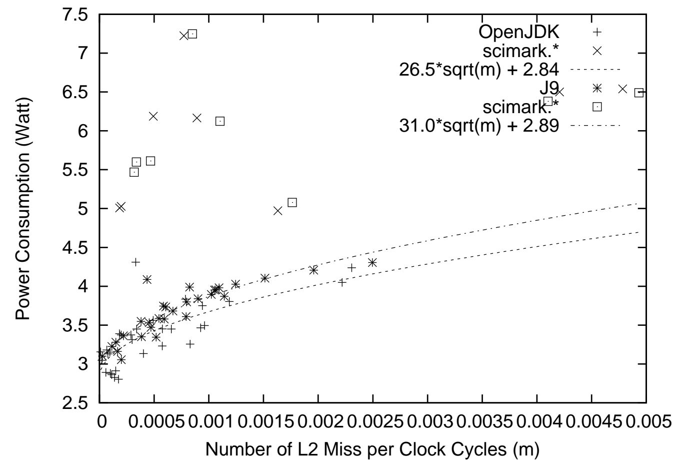

Figure 2. L2 Cache Miss and Dynamic Power Consumption

tively. SPECjvm2008, used in this paper, is targeted to client-side JVMs but also multi-threaded, to reflect the trend of (simultaneous) multi-threading and multi-core implementations of modern CPUs. [16] proposed a software based power analyzer for multi-core systems. Their model takes two input parameters, clock frequency and IPC, which is obtained from the performance counters, and predicts the dynamic power dissipation. They used SPECjvm2008 for evaluating the accuracy of their model.

[17] analyzes the workload of SPECjvm2008 on high-end desktop processors (Core 2 Duo and Core i7) in various aspects, including Base vs Peak comparisons (i. e. without and with optimizations), clock per instruction and cache/TLB miss rates, memory allocation behavior and thread scaling. Seo et. al., defined a framework to estimate the energy consumption of pervasive Java-based systems [18]. They first broke the system into components (one or more related Java classes). The energy cost of each component is further divided into computational and communication parts. The former comprises of costs of bytecode interpretation, native method invocations and monitor operations, while the latter is defined by the UDP operation parametrized by the data size. [19] presents power consumption analysis of Atom-based a mobile Internet device (MID). The options examined include C6 Deep Power Down State, video formats (data rates), Hyperthreading and hardware codec acceleration.

SPECpower_ssj2008 is a benchmark from SPEC [20]. The part of its name, ssj, indicates that it is designed to evaluate

the power and performance of server-side Java application. Its workload is derived from another benchmark from SPEC, SPECjbb2005 [21] with significant modifications. EnergyBench is a benchmark suite from EEMBC to evaluate the power consumption of embedded devices [22].

V. Conclusion

In this paper, we presented a case study in performance and power consumption analysis of JVMs with SPECjvm2008. We used two JVM implementations, OpenJDK and J9, running on an Atom-based netbook with Ubuntu operating system. We observed significant differences in relative performance between two JVMs which also varied from workload to workload. The power consumption of both JVMs for the same workload is relatively similar. Therefore, the power-performance efficiency is also close to the relative performance. In other words, the choice of JVM for the workload is important not only for the performance but also for the power efficiency.

The relative performance and power consumption of single-thread executions, again, change significantly among workloads and between JVMs. With few exceptions, the loss in performance is more than the savings in the power consumption. When the clock frequency is lowered to 800MHz (50% of the maximum frequency), the relative performance is also around 50% for most benchmarks with two exceptions. scimark.fft and scimark.lu on both JVMs are significantly faster than 50% while xml.validation on J9 is only 37% of 1.6GHz. It was observed that the power consumptions of benchmark programs (except scimark) fit well with linear and square root functions of L2 reference and L2 miss rates, respectively.

The future work include as follow. As mentioned above, scimark benchmark programs have different power consumption characteristics than other benchmark programs in SPECjvm2008. In this paper, we have only measured retired microoperations and L2 reference and miss rates. We plan to measure more workload parameters such as the floating point operation rates and bus utilization. We also plan to identify the part of benchmark programs as well as JVM implementations, which dominate the power consumption of the workloads. The platform used in this paper, an Atom-based netbook is a low-end model in today's standard and use of more powerful machines, such as many-core CPUs with larger caches, is another direction for our future work.

References

[1] Tim Lindholm and Frank Yellin, "The Java(TM) Virtual Machine Specification (2nd Edition)", Addison-Wesley Professional, 1999

[2] SPECjvm2008, http://www.spec.org/jvm2008/ .

[3] OpenJDK, http://openjdk.java.net/

[4] developerWorks : Java™technology :, http://www.ibm.com/developerworks/java/

[5] "SPECjvm2008 Run and Reporting Rules, " http://www.spec.org/jvm2008/docs/RunRules.html .

[6] SPEC CPU95, http://www.spec.org/cpu95/

[7] "MP3 library for the Java Platform," http://www.javazoom.net/javalayer/javalayer.html

[8] "Java SciMark 2.0," http://math.nist.gov/scimark2/

[9] http://www.intel.com/technology/atom/

[10] Electronic Educational Devices, https://www.wattsupmeters.com/

[11] Hitoshi Oi, "Local Variable Access Behavior of a Hardware-Translation Based Java Virtual Machine," in *Journal of Systems and Software*, pp2059–2068, Vol. 81, Issue 11, 2008. Elsevier.

[12] "Enhanced Intel SpeedStep Technology - How To Document," http://www.intel.com/cd/channel/reseller/asmo-na/eng/203838.htm

[13] "Linux kernel CPUfreq subsystem," http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html

[14] Oprofile, http://oprofile.sourceforge.net/news/

[15] Hadi Esmaeilzadeh, et. al., "Power and Performance of Native and Java Benchmarks on 130nm to 32nm Process Technologies," Sixth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS 2010), Saint-Malo, France.

[16] Shinan Wang, Hui Chen and Weisong Shi, "SPAN: A software power analyzer for multicore computer systems," in *Sustainable Computing: Informatics and Systems*, Vol. 1, Issue 1, March 2011, pp23-34.

[17] Kumar Shiv, et. al., "SPECjvm2008 Performance Characterization," in *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pp17–35, 2009.

[18] Chiyoung Seo, Sam Malek and Nenad Medvidovic, "Estimating the Energy Consumption in Pervasive Java-Based Systems," in *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, March 2008, pp243–247.

[19] Petter Larsson, "Power Efficiency - Analysis and SW Development Recommendations for Intel Atom based MID platforms," White Paper, Intel Corporation, March 2009.

[20] SPECpower_ssj2008, http://www.spec.org/power_ssj2008/ .

[21] SPECjbb2005, http://www.spec.org/jbb2005/ .

[22] EnergyBench, http://eembc.org/benchmark/power_sl.php