This paper appears in the 2017 IGSC Track on Contemporary Issues on Sustainable Computing

Energy Consumption in Java: An Early Experience

Mohit Kumar
Mobile and Internet Systems Laboratory
Wayne State University
Detroit, USA
Email: mohitkumar@wayne.edu

Youhuizi Li
Key Laboratory of Complex
Systems Modeling and Simulation
Hangzhou Dianzi University, China
Email: huizi@hdu.edu.cn

Weisong Shi
Mobile and Internet Systems Laboratory
Wayne State University
Detroit, USA
Email: weisong@wayne.edu

***Abstract*—There has been a 10,000-fold increase in performance of supercomputers since 1992 but only 300-fold improvement in performance per watt. Dynamic adaptation of hardware techniques such as finc-grain clock gating, power gating and dynamic voltage/frequency scaling, are used for many years to improve the computer's energy efficiency. However, recent demands of exascale computation, as well as the increasing carbon footprint, require new breakthrough to make ICT systems more energy efficient. Energy efficient software has not been well studied in the last decade. In this paper, we take an early step to investigate the energy efficiency of Java which is one of the most common languages used in ICT systems. We evaluate energy consumption of data types, operators, control statements, exception, and object in Java at a granular level. Intel Running Average Power Limit (RAPL) technology is applied to measure the relative power consumption of small code snippets. Several observations are found, and these results will help in standardizing the energy consumption traits of Java which can be leveraged by software developers to generate energy efficient code in future.**

***Index Terms*—ICT, RAPL, Java, Energy efficiency**

I. Introduction

Energy efficiency of Information and Communications Technologies (ICT) systems have doubled roughly every nineteen months since the invention of the first computer in 1946[1]. However, the growth in ICT systems over the years has outpaced the improvement in energy efficiency due to the affordable price and availability. According to International Telecommunication Union (ITU), two-thirds of the world's population lives in an area covered by the mobile broadband network, and 95% of the population has access to a mobile network which has resulted in an almost same number of mobile-cellular subscriptions as the human population. Each of the personal computers generates a ton of carbon dioxide every year[2]. Today, ICT amounts for 10% of the world energy[3] and 3% of the overall carbon footprint[4], which will keep on growing in future.

ICT systems are driven by hardware and software. Most of the green IT initiative concentrates on the hardware part. Earlier works on energy efficiency proposed different hardware technologies like low-power circuits, chip multiprocessors, fine grain clock gating, power gating and dynamic voltage/frequency scaling[5]–[9]. The amount of electricity consumed by different hardware components has significantly reduced over the years. However, a little has been done to improve the energy efficiency of software. An energy-aware software that can optimize execution time is also helpful in making the ICT systems more energy efficient. Software energy savings are considered to be greater than the power savings in hardware, but they are harder to achieve[10] as there is no technology that can ascertain the energy consumption of all the components in any of the ICT systems. Without accurate energy consumption measurements, it's not possible at the software level to optimize the energy efficiency.

In this paper, we undertake an early step to evaluate the energy efficiency from the perspective of programming language. Java is one of the most common languages in the ICT systems. Our paper evaluates energy consumption of data types, operators, control statements, exceptions, and objects in Java, and it will help software developers in writing energy efficient code in future. The key findings of this paper include:

- Primitive data types are more energy efficient than their wrapper classes. *long* is the most energy efficient primitive data type.
- Defining a static variable causes 50% more energy consumption as compared to defining an instance variable. Short circuit operator has better energy efficiency if the most common case is declared first.
- String concatenation operator consumes less energy than its counterpart StringBuffer and StringBuilder classes' *append* method.
- Using method call in statements, e.g., in for loop termination expression, doesn't always have more overhead than using a variable. Try-catch block has no cost if no exception is thrown.
- IntellijI IDEA is more energy efficient than Eclipse and NetBeans when installed in the default configuration.

The rest of the paper is organized as follows. Section 2 provides a background for measurement and evaluation of energy efficiency and discusses related work in the field. Section 3 describes the energy consumption measurement setup. Section 4 investigates the energy consumption traits in Java language, and Section 5 sheds light on the energy consumption of different Integrated Development Environments(IDEs). Section 6 summarizes all observations, and Section 7 concludes the paper and discusses the future work in energy efficient software.

978-1-5386-3470-7/17/$31.00

TABLE I
RAPL DOMAINS

Domain	Component
Package	CPU package
PP0	All cores and caches
PP1	GPU
DRAM	DRAM

II. BACKGROUND AND RELATED WORK

Energy efficiency plays an important role in reducing the carbon footprint of ICT systems. In this section, we will first introduce energy-related terms that help us understand the energy consumption of a system. Then, we will look at different works that have been done in software energy efficiency.

Energy is the amount of work that has been done to move an object through a distance of one meter using a force of one Newton. Energy consumption is the amount of energy required to finish a piece of work. The SI unit of energy is Joule. Power is the amount of energy consumed per unit time, and its unit is Watt.

Intel introduced Running Average Power Limit (RAPL) feature with Sandy Bridge for measuring the energy consumption of onboard hardware components. It provides energy consumption information of different CPU-level components listed in Table I. It uses a software power model which estimates the energy consumption by leveraging hardware performance counters. A user can configure and read RAPL information through Mode Specific Registers in privileged kernel mode.

Software energy efficiency has been researched for past few years. Sorting algorithms energy consumption is examined in [11] for embedded and mobile environments. Quality contracts that express dependencies between software and hardware components are used in [12], [13] for energy efficiency of software systems. The impact of languages, compiler optimization and implementation choices on Fast Fourier Transform, Linked List Insertion/Deletion and Quicksort program is examined in [14]. SEEDS and Chameleon frameworks are introduced in [15] and [16] for automating code-level changes and optimizing Java applications. These frameworks can select the most efficient collection for improving the energy efficiency of an application. Java thread management constructs - explicit thread creation, fixed-size thread pooling, and work stealing - relation to energy consumption is explored in [17]. Michanan et al. study the change in the energy efficiency of software by using different classes that implement the same interface in [18]. In their study, dynamic data structures have shown to save energy of at least 16.95% and up to 97.50% in software applications. They use machine learning tools to select the right data structure for the right workload and modify the classes for adaptive green data structures which resulted in better energy efficiency. Java collections are studied in terms of energy efficiency in [19] and [20]. In their study, selection of wrong collection type has shown 300% more energy consumption than the most efficient type. Application programmers are shown to be aware of software energy consumption problems in [21]. Our paper differs from the studies described above as none of them inspects Java on data type, operator, control statement, or exception level.

TABLE II
SYSTEM SPECIFICATION

System Component	Configuration
CPU	Intel(R) Xeon(R) E3-1270 v5
Socket	1
Number of cores	4
Number of threads	8
Microarchitecture	amd64
Kernel	Linux 4.4.0-78-generic
OS	Ubuntu 16.04.2 LTS
Memory	16GB of DDR4 2133 MHz
L1 cache	32KB I-cache, 32KB D-cache
L2 cache	256KB
L3 cache	8192KB
JDK	OpenJDK 64-Bit Server VM
JDK build	25.131-b11
JDK version	1.8.0_131

III. ENERGY CONSUMPTION MEASUREMENT SETUP

We leveraged Intel(R) Xeon(R) E3-1270 v5 server to conduct the experiments. The configuration of the system is presented in Table II. The CPU governor is set to *powersave* mode, in which CPUs run at the lowest possible frequency to avoid overheating problems. JVM runs with the default multi-thread parallel garbage collector and just-in-time(JIT) compilation. JIT compilation helps in improving the performance of a Java application by compiling at runtime. Java heap is used to store the objects during the application execution, and its size greatly influences the performance of the application. For 16GB of the memory of our system, the initial heap size is set to 262MB, and the maximum heap size is set to 4GB. Hyper-threading and Turbo Boost is enabled. Hyper-threading enables running of multiple threads on a core, which ultimately results in efficient use of core resources. Turbo Boost technology allows a core to runs faster than the specified operating frequency when other cores are operating below than their specified power, current, and temperature limits.

Energy consumption is logged using an RAPL *C* script, which reports the energy measurement of Package, PP0, and DRAM domains every second. These data are stored in a list during the application run time and logged to a text file as soon as the execution stops for further analysis.

IV. ENERGY CONSUMPTION TRAITS

In this section, we explored different code snippets to evaluate their energy consumption characteristics. For better estimation, we iterated each code ten times. We logged package, core, and DRAM active energy but only presented package

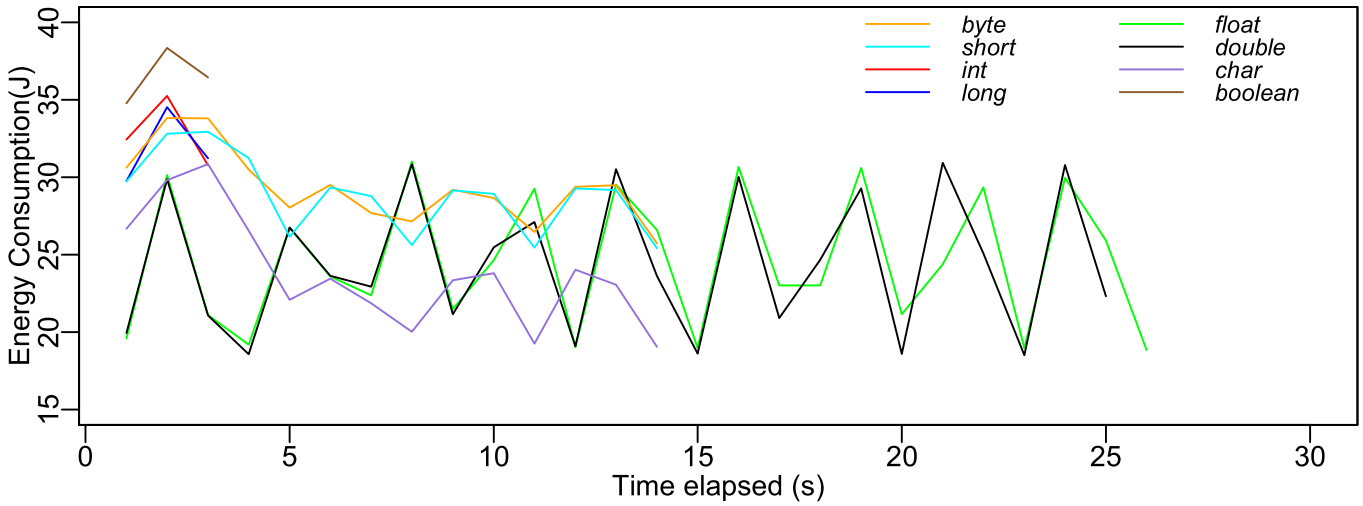


Fig. 1. Primitive data types package energy consumption.

```java
byte a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 1. byteVar.java

```java
 short a=0;

 for(int i=0;i<2000000000;i++){
  a+=1;
 }
```

Code 2. shortVar.java

```java
int a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 3. intVar.java

```java
long a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 4. longVar.java

energy consumption values as the core energy measurements were almost same to the package and the DRAM energy measurements were very low in all cases.

A. Variables

We compared all of the primitive data types - *byte*, *short*, *int*, *long*, *float*, *double*, *char*, and *boolean* - in terms of energy efficiency in Fig. 1. The codes for these data types is shown in Code 1, 2, 3, 4, 5, 6, 7, and 8. The main function and class declaration have the same syntax for all the programs. For each data type, we declared a variable and executed addition assignment statement two billion times. The *int* and *long* completed the execution in less than 5 seconds which took an energy of 23 and 21 joules respectively. The most possible reason for the *int* higher energy consumption compared to the *long* can be the 64-Bit JVM. The *float* and *double* took around 600% more time for the same operation and consumed 639 and 610 joules of energy respectively. The *byte*, *short*, and *char* took almost half of the time and energy than the *float* and *double*. According to these results, a programmer should always try to avoid the *byte*, *short*, *float*, *double*, and *char* types for better application performance and energy efficiency.

Variables in Java can be declared as instance and static

```java
float a=0.0f;

for(int i=0;i<2000000000;i++){
 a+=1.0f;
}
```

Code 5. floatVar.java

```java
double a=0.0;

for(int i=0;i<2000000000;i++){
 a+=1.0;
}
```

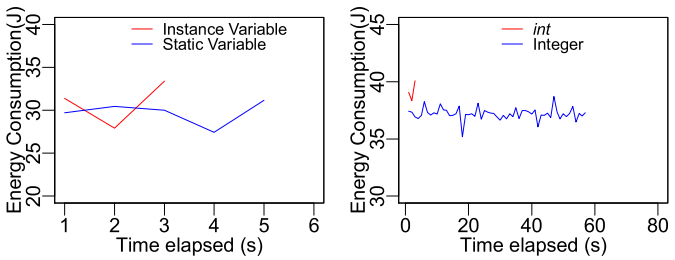
Code 6. doubleVar.java

```java
 char a=0;

 for(int i=0;i<2000000000;i++){
  a+=1;
 }
```

Code 7. charVar.java

```java
 boolean a=true;

 for(int i=0;i<2000000000;i++){
  a=true;
 }
```

Code 8. booleanVar.java



(a) (b)

Fig. 2. Instance and static variable (a) and int and Integer (b) package energy consumption.

variables. Instance variables are stored in the stack and have very fast performance. They can be changed in their specific scope during the program execution. Static variables are stored in the heap and have a global scope. For the instance and static variable, we executed addition assignment statement two billion times in Code 9 and 10. The results of the energy consumption measurements for both cases are shown in Fig. 2a. The static variable caused 50% more energy consumption and took 60% more time to execute as compared to the instance variable. The slower heap memory may be the reason for the static variable behavior.

Wrapper classes in java are used to convert a primitive data type into an object and vice-versa. Integer wrapper class is used to convert primitive data type *int* into an object. It is used in cases like storing *int* values in a collection as they only accept an object as values. *int* is considered to be faster as it is a primitive data type, whereas Integer class object carries extra overhead due to functions calls and object initialization. For testing, we initialized a variable of both types to zero and executed addition assignment statement for both types two billion times as shown in Code 11 and 12. As expected, *int* variable performed much better in performance and energy efficiency. As shown in Fig. 2b, *int* code finished in 5 seconds but Integer code took almost 60 seconds. The Integer code consumed a whopping 2119 joules of energy while the *int* code consumed 117 joules. Therefore, developers should always prefer primitive data types over their wrapper classes.

```
int i=0;

for(i=0;i<2000000000;i++){
 i+=1;
}
```

Code 9. instanceVar.java

```
public class staticVar {
static int i;
public static void main(String args[]){

 for(i=0;i<2000000000;i++){
  i+=1;
 }

 System.out.println("Done");

 }
}
```

Code 10. staticVar.java

```
int a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 11. intInteger.java

```
Integer a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 12. integerInt.java

```
int a=0;

for(int i=0;i<2000000000;i++){
 if(a>-1 || a>1 || a>1)
  a+=1;
}
```

Code 13. circuitLeftOr.java

```
int a=0;

for(int i=0;i<2000000000;i++){
 if(a<-1 || a<-1 || a>-1)
  a+=1;
}
```

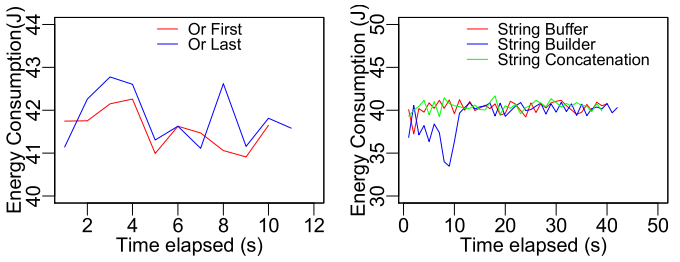
Code 14. circuitRightOr.java

B. Short Circuit operator

|| and && operators are called short-circuit operators in Java. When the first operand of the && operator evaluates to false, the result of the whole condition is false, and when the first operand of the || operator evaluates to true, the result of the whole condition is true. We compared the two cases where the true operand of the || operator was in the first and last position. As shown in Code 13 and 14, we created a for loop with two billion iterations, with if condition having three operands for the || operator. For Code 13, the result is returned after the evaluation of the first operand. For Code 14, the result is returned after evaluation of the last operand. The first and second cases are represented as Or First and Or Last in Fig. 3a, respectively. The Or Last case consumed 10% more energy than the Or First case, thus it is better to always put those cases first in short circuit operators that are most common. The execution time of the Or Last case was higher than the Or First case by half of a second. The && operator showed the same behavior.

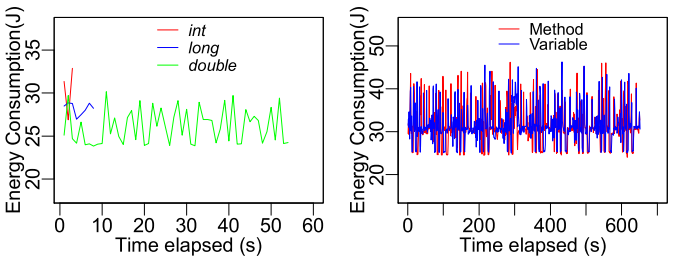
C. String

String is immutable and can't be altered once created in Java. Applying different methods on a String doesn't change the original string. For example, String.substring(int beginIndex) returns a new string that is a substring of the original



(a) (b)

Fig. 3. First operand as true and last operand as true in || operator expression (a) and StringBuffer, StringBuilder and concatenation operator (b) package energy consumption.



(a) (b)

Fig. 4. Initialization expression (a) and termination expression (b) package energy consumption.

```
StringBuffer str=new StringBuffer("Iteration ");

for(int i=0;i<100;i++){
 System.out.println(str.append(i));
}
```

Code 15. stringBuffer.java

```
StringBuilder str=new StringBuilder("Iteration ");

for(int i=0;i<100;i++){
 System.out.println(str.append(i));
}
```

Code 16. stringBuilder.java

```
String str="Iteration ";

for(int i=0;i<100;i++){
 System.out.println(str+i);
}
```

Code 17. stringConcatenation.java

```
int a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 18. loopInt.java

```
int a=0;

for(long i=0;i<2000000000;i++){
 a+=1;
}
```

Code 19. loopLong.java

```
int a=0;

for(double i=0;i<2000000000;i++){
  a+=1;
}
```

Code 20. loopDouble.java

string. String supports concatenation operator (+) which helps in combining two or more Strings. The concatenation operator is resolved at compile time if all the Strings that need to be combined can be resolved at compile time. However, if an expression can't be resolved at compile time, the concatenation operator will execute at runtime and causes extra overhead[22]. StringBuffer and StringBuilder classes have the same function *append* for concatenation and allow the creation of mutable strings. The difference between them is that the StringBuffer is synchronized and can be accessed by several threads at the same time whereas the StringBuilder is not synchronized and should be accessed by a single thread. We attempt to find out whether runtime string concatenation is more efficient than the *append* method of StringBuffer and StringBuilder.

For the StringBuffer and StringBuilder, we initialized objects with value "Iteration" and then we called *append* function 100 times on the same objects passing loop control variable as a parameter. For the concatenation operator, we initialized the string variable to "Iteration" and then used the concatenation operator to append loop control variable 100 times. The codes for the StringBuffer, StringBuilder, and concatenation operator is shown in Code 15, 16, and 17. When we looked at the results, the concatenation operator was found to be the fastest and most energy efficient. The StringBuilder took 10% more time whereas the StringBuffer took 4% more time than the concatenation operator. For energy consumption, they followed the same trend which is shown in Fig. 3b. The StringBuilder consumed less energy at first, but the energy increased as it ran and reached the maximum value. These results show that the concatenation operator is better than the StringBuffer's and StringBuilder's *append* method even when it is compiled at run time.

D. Loop

Loops are frequently used to execute a block of statements or a single statement several times. For our analysis, we first changed the initialization variable to a different data type in a for loop - *int*, *long*, and *double* -as shown in Code 18, 19, and 20, and then we executed addition assignment statement two billion times in each code. The results for these cases are shown in Fig. 4a, which are slightly different than that of the primitive data type in the variable sub-section. The *int* is the fastest instead of the *long*, and the *double* is the slowest.

We also examined two different ways of initializing the termination expression: using a variable and using a method

call. In both cases, we first initialized an ArrayList and added two billion values to it. Then, in the method case, we used size method in for loop termination expression, and in variable case, we first stored the size of the list in an integer variable and then used that variable in for loop termination expression to avoid multiple calls to the size method. The codes for the method and variable termination is shown in Code 21 and 22 respectively. Surprisingly, the energy consumed in the variable termination case was higher by 100 joules with same execution time for both cases as shown in Fig. 4b. Compiler optimization techniques like memoization can be the reason for lower energy consumption in method case.

E. Exception

Exceptions help us in tracking down the error. Java uses try-catch blocks to handle exceptions. First, we examined whether

```
ArrayList loop=new ArrayList();

for(int i=0;i<2000000000;i++){
 loop.add(i);
}

for(int i=0;i<loop.size();i++){
 loop.get(i);
}
```

Code 21. loopMethodTerminate.java

```
ArrayList loop=new ArrayList();

for(int i=0;i<2000000000;i++){
 loop.add(i);
}

int size=loop.size();
for(int i=0;i<size;i++){
 loop.get(i);
}
```

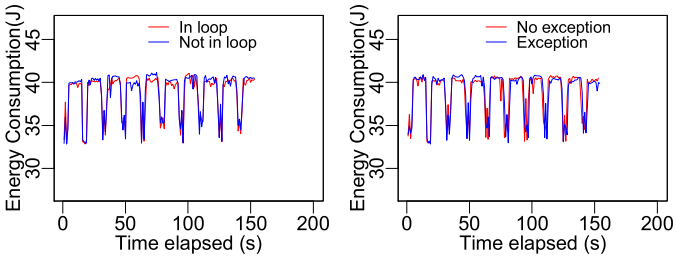
Code 22. loopVarTerminate.java

```
 int a=1,b=1;

 for(int i=0;i<2000000000;i++){
  try{
  a=a/b;
  }catch(Exception E){
   System.out.println("Denominator can't be
       zero!!!");
  }
 }
```

Code 23. exceptionInLoop.java

```
 int a=1,b=1;
 try{
 for(int i=0;i<2000000000;i++){

  a=a/b;

 }
 }catch(Exception E){
  System.out.println("Denominator can't be
      zero!!!");
 }
```

Code 24. exceptionOutLoop.java



(a) (b)

Fig. 5. Try-catch in loop and outside loop (a) and Exception in loop and no exception in loop (b) package energy consumption.

```
int a=1,b=1;

for(int i=0;i<2000000000;i++){
 try{
 a=a/0;
 }catch(Exception E){
  a=a/b;
 }
}
```

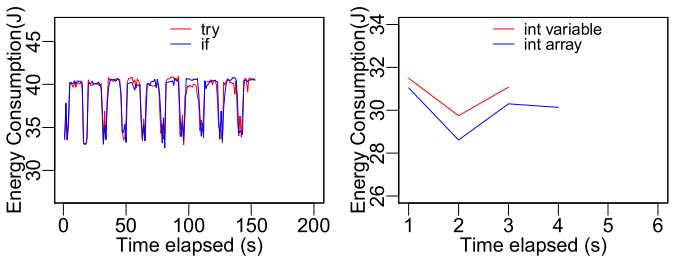
Code 25. exceptionThrownLoop.java

```
int a=1,b=1;

for(int i=0;i<2000000000;i++){
 try{
 a=a/b;
 }catch(Exception E){
  a=a/b;
 }
}
```

Code 26. exceptionNotThrown.java

a try-catch block has any associated cost when it is used in a program with no exception thrown. We compared the energy consumption for multiple calls of the try-catch block inside a loop in Code 23 with a single call to a try-catch block outside the loop in Code 24. The first case resulted in two billion executions of a try-catch block while the second case resulted in the single execution of a try-catch block. However, when we compared the execution time and energy efficiency of both codes, it was found to be almost the same. The Code 24 consumed less than 1% more time and energy than the Code 23. The energy consumption data of the two cases is shown in Fig. 5a.

Executing a catch block has major overhead as it involves reading the stack for the thrown exception. We measured the impact of the catch block by running two cases - exception thrown and no exception thrown - using the Exception class. All exceptions in Java are subtypes of the Exception class. In the exception thrown case, we put a try-catch block inside a for loop which iterated two billion times. For each iteration, an exception was thrown due to division by zero as shown in Code 25. Then in Code 26, we wrote same statements with no exception thrown. The results for both cases were the same as shown in Fig. 5b. The results may differ for a different exception type which we haven't explored at this time.



(a) (b)

Fig. 6. Try-catch and if (a) and primitive data type *int* and *int* array (b) package energy consumption.

```
int a=1,b=1;

for(int i=0;i<2000000000;i++){
 try{
 a=a/b;
 }catch(Exception e){
  System.out.println("Denominator can't be
      zero!!!");
 }

}
```

Code 27. exceptionIf.java

```
int a=1,b=1;

for(int i=0;i<2000000000;i++){
 if(b==0)
  System.out.println("Denominator can't be
      zero!!!");
 a=a/b;

}
```

Code 28. ifException.java

```
int a=0;

for(int i=0;i<2000000000;i++){
 a+=1;
}
```

Code 29. intArr.java

```
int a[]={0};

for(int i=0;i<2000000000;i++){
 a[0]+=1;
}
```

Code 30. arrInt.java

In high-level languages, people use if condition and try-catch block interchangeably. We tried to find out which one has more overhead for the same type of error check. We divided two integers. For Code 27, we used a try-catch block whereas, for Code 28, we used if condition to check whether the denominator is zero during division. The two cases followed the same pattern in energy consumption as shown in Fig. 6a. The if case consumed five more joules and took one more second to finish the same work. Consequently, It is better to avoid the if condition for error checking and use the try-catch block.

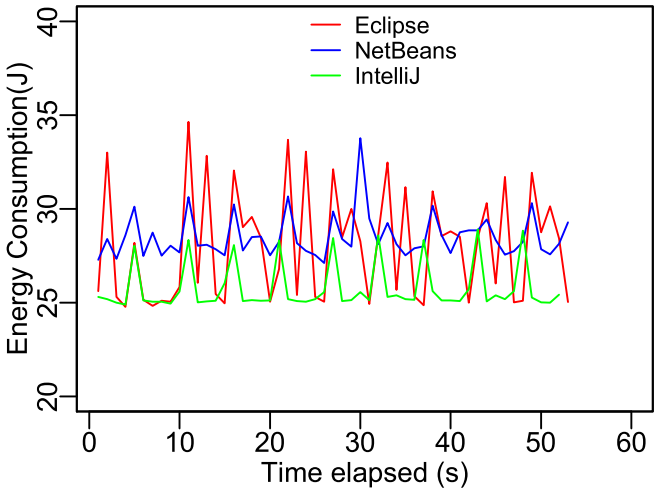


Fig. 7. Top 3 IDEs package energy consumption.

F. Array

An array can store multiple values of the same type. In Java, arrays are objects that are created dynamically. The number of variables in an array can be zero. The variables contained in an array are accessed by a non-negative integer index. We inspected how a variable and an array with one variable differs in terms of energy efficiency. For the variable case, we initialized an *int* variable to zero, and for the array case, we initialized an *int* array of size one to zero. We executed addition assignment statement two billion times for both cases as shown in Code 29 and 30. The array case was found to consume 33% more energy as shown in Fig. 6b.

V. IDEs Energy Profile

Every programmer needs an IDE to assist in writing code. An IDE selection depends on many factors like project, organization and programming skills. An IDE helps in compiling, building, debugging and testing a code in the same window. In this section, we compared the energy consumption of the top three IDE in the market that used to develop Java applications - Eclipse Neon, NetBeans 8.1, and IntelliJ IDEA Community Edition. The big difference between them is how they are developed. While Eclipse and NetBeans are managed by a vibrant community of developers, IntelliJ is managed by a single company. This helped IntelliJ to have more cohesive built-in features. We first installed each of the above-mentioned IDE with default configuration, and then we ran the Code 20 for testing. The results showed that IntelliJ IDEA is the most energy efficient IDE consuming 1340 joules of energy. Eclipse and Netbeans consumed 10% and 13% more energy than IntelliJ IDEA. All IDEs took almost same time for the code execution as shown in Fig. 7.

VI. Summary

In this section, we summarized all our observations about energy consumption in Java in Table III.

TABLE III
A SUMMARY OF THE OBSERVATIONS

Component	Observation
Variables	*long* is the most efficient primitive data type. Static variables consume 50% more energy as compared to instance variables. Primitive data types are much efficient than their wrapper classes.
Short Circuit operator	Putting most common cases first can save up to 10% energy.
String	String concatenation operator consumes less energy than StringBuilder and StringBuffer classes' *append* method both at run time and compile time.
Loop	*int* is the most energy efficient initialization variable in a for loop. Method termination expression consumes less energy than variable termination expression.
Exception	Try-catch block consumes almost no energy when no exception is thrown. Catch block execution has very low cost in case of an exception. Using an if condition for error checking causes higher energy consumption.
Array	Array consumes 33% more energy than instance variable.
IDEs	IntelliJ IDEA is more energy efficient than Eclipse and Netbeans.

VII. Conclusion & Future Work

Software energy efficiency has been still in its infancy stage as most of the energy efficiency work concentrates on hardware part. Software developers have been oblivious about the application energy efficiency for years. They do not pay attention to the energy needs of the software they develop. There are very limited standards that they can follow to write energy efficient code. This paper uncovers some of the clues at a granular level which will bolster energy efficient developing standards. We measured and compared the energy efficiency of a static variable, instance variable, primitive data types, wrapper classes, String concatenation operator, try-catch block, short circuit operators, and arrays. For IDEs, IntellijI IDEA found to be more energy efficient than Eclipse and NetBeans when installed in the default configuration. We hope that these results will help software developers to build more energy efficient Java application in future.

This paper just scratches the surface of the domain of software energy efficiency. Our future work will focus on evaluating the energy efficiency of more complex data types, inheritance, packages, interfaces, and multithreading in Java. Researchers can also take a hint from this paper to explore other programming languages for energy efficiency traits.

Acknowledgment

This work is supported in part by National Science Foundation (NSF) grant CNS-1561216.

References

[1] J. Koomey, S. Berard, M. Sanchez, and H. Wong, "Implications of historical trends in the electrical efficiency of computing," *IEEE Annals of the History of Computing*, vol. 33, no. 3, pp. 46–54, 2011.

[2] S. Murugesan and G. Gangadharan, *Harnessing green IT: Principles and practices*. Wiley Publishing, 2012.

[3] M. Mills, "The cloud begins with coal. digital power group," 2013.

[4] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *International Journal of Computer Aided Engineering and Technology*, vol. 6, no. 4, pp. 440–459, 2014.

[5] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low-power cmos digital design," *IEICE Transactions on Electronics*, vol. 75, no. 4, pp. 371–382, 1992.

[6] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, "1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos," *IEEE Journal of Solid-state circuits*, vol. 30, no. 8, pp. 847–854, 1995.

[7] T. D. Burd and R. W. Brodersen, "Energy efficient cmos microprocessor design," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 1. IEEE, 1995, pp. 288–297.

[8] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE Journal of solid-state circuits*, vol. 35, no. 11, pp. 1571–1580, 2000.

[9] J. B. Burr and A. M. Peterson, "Energy considerations in multichip-module based multiprocessors," in *Computer Design: VLSI in Computers and Processors, 1991. ICCD'91. Proceedings, 1991 IEEE International Conference on*. IEEE, 1991, pp. 593–600.

[10] J. Burr, L. Gal, R. Haddad, J. Rabaey, and B. Wooley, "Which has greater potential power impact: High-level design and algorithms or innovative low power technology?(panel)," in *Proceedings of the 1996 international symposium on Low power electronics and design*. IEEE Press, 1996, p. 175.

[11] C. Bunse, H. Höpfner, E. Mansour, and S. Roychoudhury, "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments," in *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*. IEEE, 2009, pp. 600–607.

[12] S. Götz, C. Wilke, S. Richly, and U. Aßmann, "Approximating quality contracts for energy auto-tuning software," in *Proceedings of the First International Workshop on Green and Sustainable Software*. IEEE Press, 2012, pp. 8–14.

[13] S. Götz, C. Wilke, M. Schmidt, S. Cech, and U. Aßmann, "Towards energy auto tuning," 2010.

[14] S. Abdulsalam, D. Lakomski, Q. Gu, T. Jin, and Z. Zong, "Program energy efficiency: The impact of language, compiler and implementation choices," in *Green Computing Conference (IGCC), 2014 International*. IEEE, 2014, pp. 1–6.

[15] I. Manotas, L. Pollock, and J. Clause, "Seeds: a software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 503–514.

[16] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: adaptive selection of collections," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 408–418.

[17] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 345–360, 2014.

[18] J. Michanan, R. Dewri, and M. J. Rutherford, "Predicting data structures for energy efficient computing," in *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*. IEEE, 2015, pp. 1–8.

[19] G. Pinto, K. Liu, F. Castor, and Y. D. Liu, "A comprehensive study on the energy efficiency of java thread-safe collections," *Journal of Systems and Software*, 2016.

[20] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 225–236.

[21] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 22–31.

[22] J. Shirazi, *Java performance tuning*. " O'Reilly Media, Inc. ", 2003.