A MULTI-OBJECTIVE AUTOTUNING FRAMEWORK FOR THE JAVA VIRTUAL MACHINE

by

Shuvabrata Saha

A thesis submitted to the Graduate College of
Texas State University for the degree of
Master of Science
with a Major in Computer Science
May 2016

Committee Members:

Apan Qasem, Chair

Michael Ekstrand, co-Chair

Xiao Chen

COPYRIGHT

by

Shuvabrata Saha

2016

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgement. Use of this material for financial gain without the author's express written permission is not allowed.

Duplication Permission

As the copyright holder of this work I, Shuvabrata Saha, authorize work, in whole or in part, for educational or scholarly purposes only.

DEDICATED to

my MOTHER and FATHER

and

my FRIENDS

ACKNOWLEDGEMENTS

I would like to express my deep-felt gratitude to my advisor Dr. Apan Qasem, department of computer science at Texas State University for his advice, encouragement, enduring patience and constant support. He was never ceasing in his belief in me and always providing clear guidance when I was lost, constantly driving me with energy.

I would like to thank my co-advisor Dr. Michael Ekstrand,department of Computer Science at Texas State University. His suggestions, comments and additional guidance were invaluable to the completion of this work.

I also wish to thank Dr. Xiao Chen, department of Computer Science at Texas State University, for being on my committee and for their constructive feedbacks.

Additionally, I want to thank the professors of department of Computer Science at Texas State University for all the hard work and dedication, providing me the means to complete my degree.

Last but not least, I would like to thank my parents for their immense support during this time of my life.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS v
LIST OF TABLES viii
LIST OF FIGURES ix
LIST OF ABBREVIATIONS x
ABSTRACT xi
CHAPTER

I. INTRODUCTION 1
- High Performance in JVM 1
- Auto-Tuning 1
- Multi-objective tuning 2
- Contribution of this Thesis 3
- Overview of the Next Chapters 3

II. RELATED WORK 4
- JVM Tuning 4
- Energy Efficiency 5
- Statistical Model 5

III. JTUNER FRAMEWORK 7
- Overview 7
- Java Virtual Machine 7
- Category of JVM Flags 8
- The chosen flags 9
- Search Space Construction 12

Heuristic Search 13
Top-Most Independent Probability 13
Independent Identical probability 13
Statistical Modeling 13
Linear Regression 13
Decision Tree 14
Analysis and Recommendation 14

IV. EXPERIMENTAL RESULTS 18
Environment and Setup 18
Basic Matrix Computation analysis 18
Benchmark Configuration 19
Lenskit Configuration 21
Test Result of Matrix Computation 22
Test Result of SPECjvm Benchmark 29
Test Result of DaCapo Benchmark 35
Test Result of Lenskit tool 41
Algorithms Comparison 48
Summary 49

V. CONCLUSION AND FUTURE WORK 50
Conclusion 50
Future Work 50
REFERENCES

LIST OF TABLES

Table **Page**

III.1 Advance JIT Flags 10

III.2 Discarded Flags 10

III.3 Discarded Flags 11

III.4 Advance Garbage Flags 11

III.5 HotSpot JVM Flags 12

IV.1 SPECjvm Benchmarks 19

IV.2 DaCapo Benchmarks 20

LIST OF FIGURES

Figure **Page**

III.1 JVM Architecture Diagram 8

III.2 Architecture Diagram 12

III.3 JVM Tuning GUI 15

III.4 Data Setup Panel 15

III.5 Sample Excel File 16

III.6 Algorithm Panel 17

III.7 Algorithm Panel 17

IV.1 Matrix Analysis 22

IV.2 Time Analysis For Matrix Tasks 23

IV.3 Power Analysis For Matrix Tasks 24

IV.4 SPECjvm Analysis 29

IV.5 Time Analysis For SPECjvm Tasks 30

IV.6 Power Analysis For SPECjvm Tasks 31

IV.7 DaCapo Analysis 35

IV.8 Time Analysis For DaCapo Tasks 36

IV.9 Power Analysis For DaCapo Tasks 37

IV.10 Lenskit Analysis 41

IV.11 Time Analysis For Lenskit Tasks 42

IV.12 Power Analysis For Lenskit Tasks 43

LIST OF ABBREVIATIONS

TIP Top-Most Independent Probability

IID Independent Identical Probability

JVM Java Virtual Machine

JRE Java Runtime Environment

JDK Java Development Kit

ABSTRACT

Due to inherent limitations in performance, Java was not considered a suitable platform for for scalable high-performance computing (HPC) for a long time. The scenario is changing because of the development of frameworks like Hadoop, Spark and Fast-MPJ. In spite of the increase in usage, achieving high performance with Java is not trivial. High performance in Java relies on libraries providing explicit threads or relying on runnable-like interfaces for distributed programming. In this thesis, we develop an autotuning framework for JVM that manages multiple objective functions including execution time, power consumption, energy and perfomance-per-watt. The framework searches the combined space of JIT optimization sequences and different classes of JVM runtime parameters. To discover good configurations more quickly, the framework implements novel heuristic search algorithms. To reduce the size of the search space machine-learning based pruning techniques are used. Evaluation on recommender system workloads show that significant improvements in both performance and power can be gained by fine-tuning JVM runitme parameters.

I. INTRODUCTION

High Performance in JVM

It has been while Java in High Performance Computing (HPC) was not promoted because of its performance. The scenario is improving through some projects such as Java Fast Sockets (Taboada et al., 2008), Fast-MPJ (Taboada et al., 2012), MPJ Express (Baker et al., 2006) that aim to facilitate fast message-passing without JNI (Gordon, 1998) overhead. Hadoop (White, 2012) and Spark (Karau et al., 2015) have been very instrumental in this rise. These efficient features combined with the ease of programmability have made Java more popular in the HPC world. HPC applications built in Java are being increasingly used in other fields such as data mining, machine learning, particle physics, bioinformatics, and finance.

In spite of the increase in usage, achieving high performance with Java is not trivial. As a whole, the compiler has to map the total constructs to specific architectures. Hence achieving better runtime performance of HPC applications built over compilers such as JVMs requires further efforts.

Auto-Tuning

In order to make the compilers effective for HPC applications, different methodologies have been implemented by either manually or automatically to achieve the improvement of running time. Manual JVM system optimization is quite difficult due to the complexity of different combination of JVM parameters. There is an additional set of things to tune: the runtime parameters i.e. the garbage collector settings. Although HotSpot JIT (Paleczny et al., 2001) is quite smart and sophisticated to make some intelligent autotuning, it is still difficult to deduce the best optimization sequence for improved runtime performance. This hindrance drives us to research JVM auto-tuning process to find the best

parameters to maximize the performance in HPC. Auto-tuning techniques have been extensively investigated by researchers in the performance and computation for the last few decades. Still it remains a challenging task due to the individuality of the algorithm.

Now-a-days the application developers are now investing significant time to tune their codes for the current systems. To improve the code parts, the tuning process has been applied to accumulate data and identify the pivotal code regions. In spite of a number of autotuning researches throughout the world, still relatively few scopes have been achieved. Actually, the algorithms and data structures are not enough to boost the program performance. Rather, coding and compiler parameters also play an important role in alter. The parameters including frequency and size of messages, minimum number of iterations are required for parallel execution of a loop. There are other pivotal machine parameters such as cache size,memory bandwidth, communication costs and overhead which may also make an impact. Furthermore, some of the parameters must be reassessed for different machine porting.

Multi-objective tuning

In the performance tuning world, our goal is to improve energy and running time using different techniques such as source-level optimizations done by manually or automatically. Earlier, the developers were concerned with their running time complexity of their applications. From the algorithmic point of view, they also used to focus on memory space. For unmanaged languages, a lot of works have been standardized for this purpose since a long time. But these are not strongly found in modern languages like Java, C# etc. Because of intensity of workload performance, there has not been much work. But as time passed on and the energy also becomes pivotal item like performance, the fine-tuning optimization parameters are going to be more demanding. But this process is quite complex and time consuming because we need to deal with all the combinations of

optimization parameters. To mitigate this time-consuming problem, we need a method to reduce the search space. Although the method is quite smart and sophisticated in some extends, still it is quite tough to explain the best optimization sequence.

Contribution of this Thesis

Considering all these issues, we propose an autotuning framework for JVM. The autotuning process generates combination of the flags for the next set of optimizations and iterates until satisfactory performance is achieved. The proposed framework encompasses features like minimize power, execution time, energy and executing time at a power capacity etc. The space optimization sequences is large which makes exhaustive search infeasible. To discover good sequences more quickly, this framework will implement different heuristic search algorithms. To reduce the size of the search space, the framework will employ machine-learning based pruning techniques, including identifying groups of optimizations with negative interaction and positive interaction. The proposed framework is evaluated with real-world data sets for recommender system framewok named Lenskit (Ekstrand et al., 2011). Additionally, we will evaluate the efficiency of the proposed framework on a set of Java benchmark suites.

Overview of the Next Chapters

In Chapter II relevant studies of different auto-tuning methods are presented and describes the variety of ideas and technologies used by researchers. In Chapter III, we also categorize the total number of flags and choose the pivotal flags which are fully mentioned. Here, we also present a detailed discussion on the proposed design and implementation of the multi-objective tuning method. Then we present experimental results as well as analyses in Chapter IV. Finally, Chapter V consists of the concluding remarks and future work.

II. RELATED WORK

JVM Tuning

Many JVM optimization strategies have been proposed which are also sophisticated in some extends. (Fernando et al.) has been improved by all kinds of parameters and options supported by the JVM and can be specified at the time the Java runtime environment is started. The strategy behind auto-tuner is to classify the JVM flags into a flag hierarchy tree structure to resolve the dependency among the JVM flags. Open-tuner framework automates the combination of JVM flags in OpenJDK HotSpot VM. (Jayasena et al., 2015) show that enabling the HotSpot auto-tuner to search through the configuration space with the support of the flag hierarchy helps it to converge to a local optimum more aggressively where without this, it takes more time to output the same level of performance improvement for a given benchmark. Here, the Compilation Rate of the Tuned configuration shows a significant improvement over the default configuration. (Jantz and Kulkarni, 2013a) examines the properties of single-tier and multi-tier JIT compilation strategies that can enable existing and future VMs to realize the best program performance on modern machines. The compilers aggressively compile more program methods quickly and significantly benefits program performance especially for slower JIT compilers. In a nutshell, it proves that a tiered compilation policy, although complex to implement, greatly alleviates the more impact and early JIT compilation of programs on modern machines. (Singer et al., 2011) propose MRJ, a MapReduce Java framework for multi-core architectures. Using memory management autotuning techniques based on machine learning the authors have achieved MRJ performance within 10 percent of optimal on 75 percent of benchmark tests. Michael (Jantz and Kulkarni, 2013b) examine the phase selection related behavior of optimizations, and assessing and improving the

effectiveness of existing heuristic solutions. They try to minimize the compilation overhead using parallel computing resources in modern multiprocessors. The authors modified the HotSpot compiler to provide command-line flags for most optimization phases, and factored out the analysis calculation so that it is computed regardless of the optimization setting. The goal of effective phase selection is to find and disable optimizations with negative effects for each program region. Basically the previous papers try to mitigate the compilation load and try to maximize performance using JVM hierarchy, parallel processing technique. However, our work leads to the multiple classes of JVM parameters where the final destination is to optimize multiple objectives using machine learning models. Here, power and execution time will also be considered to address our work improvement.

Energy Efficiency

Now-a-days, energy efficiency is also a burning factor in the development world. (Vega et al., 2012) has proposed to make proper placement of threads to decrease power consumption. Power aware scheduling has been applied with deadline constraints in (Kim et al., 2007) which is applicable for multicore systems (Bautista et al., 2008). The multicore system also focuses resource scheduling (Merkel et al., 2010) and variation-aware application scheduling (Teodorescu and Torrellas, 2008). Besides, (Curtis-Maury et al., 2008) have proposed prediction models to gain energy efficiency which has also been introduced in (Contreras and Martonosi, 2005).

Statistical Model

A number of models have been approached in auto-tuning world. Single-run feedback and mvc design is used in (Huang et al., 2005). Based on input sensitivity, there are some autotuning algorithmic choices (Ding et al., 2015). Milepost GCC (Fursin et al., 2011) has been proposed as the first

publicly-available open-source machine learning-based compiler. The tuning framework has been developed to predict the optimal configuration which perform one percent of the best performance of any single configuration for the same set of applications (Liao et al., 2009).

III. JTUNER FRAMEWORK

Overview

Here, our framework will calculate multi-objects like power, time from each Java command using advanced options flag. Gathering all the combinations, we can make a statement which flag combination will be the most optimized one.

Java Virtual Machine

A Java Virtual Machine (JVM) is a virtual computing device by which a computer can run a Java program. Its instance is an implementation of a process that converts java program into java bytecode. And its implementation is a computer program of such a specification that omits implementation details that are not essential to ensure interoperability. That means it discards those which are necessarily constrain implementers. Having a single specification ensures all implementations are inter-operable. Java program can be run only inside some implementations of the abstract specification of JVM. To run a sample Java program, we also need to have Java Runtime Environment (JRE). JRE includes a JVM implementation along with Java Class Library implementation. Java Development Kit (JDK) is the superclass of JRE which contains tools for the programmers.

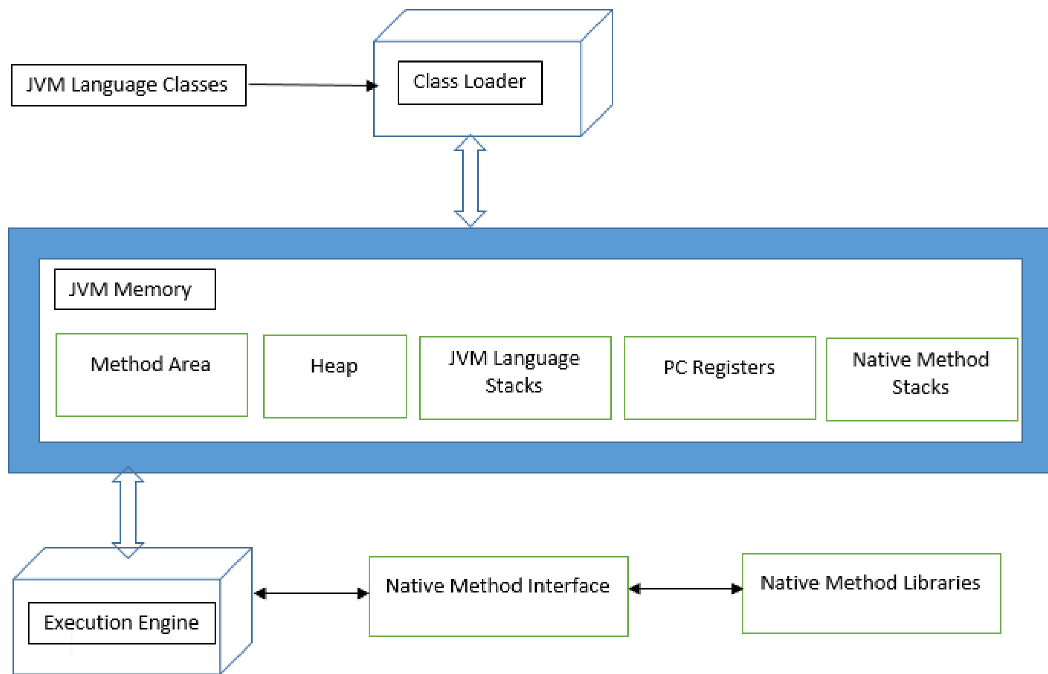


Figure III.1: JVM Architecture Diagram

The above figure describes JVM architectural diagram. Firstly, the classloader loads the class file of a sample java program. Then the method area stores the per-class structure for the methods. The structures are allocated in 'Heap' data area. Stack stores local variables and partial results. Each stack has been created at the same time as thread.

PC register contains the address of JVM instruction being executed. Finally, native method stacks contain all the native methods in the application.

From JVM memory, it goes to execution engine which contains a virtual processor, interpreter to read bytecode stream and Just-In-Time compiler to improve the performance. Then it goes to JNI(Java Native Interface) where it enables Java code running in a JVM and be called by native applications and native method libraries written in other languages such as C,C++ and assembly.

Category of JVM Flags

The java command supports a wide range of options that can be divided into the following categories:

1 Standard Options

2 Non-Standard Options

3 Advanced Runtime Options

4 Advanced JIT Compiler Options

5 Advanced Serviceability Options

6 Advanced Garbage Collection Options

Standard options for the JVM are used for checking the version of JRE, setting the classpath. Non-standard options not guaranteed for all JVM implementations and change according to the subjects.
Advanced options for tuning specific areas of the Java HotSpot need privileged access to configure any parameter. For our proposed framework, we will consider advanced options only due to its ability of tuning JVM environment. Initially we would consider Boolean options which are used to either enable or disable a feature that is enabled by default. Boolean -XX options are enabled using the plus sign (-XX:+OptionName) and disabled using the minus sign (-XX:-OptionName).

The chosen flags

In our work, we have considered only boolean flags. In case of standard options, we haven't found any flag which can make any impact in performance part as all the flags are used for general actions. This is also applicable for non-standard options. In the advanced runtime options, advanced `JIT` and `Garbage` are the primary means of tuning the performance of the JVM which has been focus here. Printing log data or assembly code or any diagonstic output will increase the running time. So, we have discarded as follows:
LogCompilation
PrintAssembly

PrintCompilation

PrintInlining

Table III.1: Advance JIT Flags

AggressiveOpts	UseCondCardMark
BackgroundCompilation	Inline
DoEscapeAnalysis	UseSuperWord
UseCodeCacheFlushing	UseAES

Advanced Garbage: For printing options, the flags in table III.2 have been discarded.

Table III.2: Discarded Flags

G1PrintHeapRegions	PrintAdaptiveSizePolicy
PrintGCApplicationConcurrentTime	PrintGC
PrintGCApplicationStoppedTime	PrintGCDateStamps
PrintStringDeduplicationStatistics	PrintGCDetails
PrintGCTaskTimeStamps	PrintGCTimeStamps
PrintTenuringDistribution	

Changing the default values of the parameters in III.3 makes a hindrance of the computation so that the application can't be run properly:

Table III.3: Discarded Flags

AggressiveHeap	DisableExplicitGC
UseAdaptiveSizePolicy	UseSerialGC
ParallelRefProcEnabled	UseG1GC
UseConcMarkSweepGC	UseStringDeduplication
UseParallelOldGC	UseParNewGC
ExplicitGCInvokesConcurrent	

Table III.4: Advance Garbage Flags

AlwaysPreTouch	ParallelRefProcEnabled	UseParallelGC
CMSClassUnloadingEnabled	ScavengeBeforeFullGC	UseSHM
CMSScavengeBeforeRemark	UseGCOverheadLimit	UseTLAB
UseCMSInitiatingOccupancyOnly	DisableExplicitGC	UseNUMA
ExplicitGCInvokesConcurrentAndUnloadsClasses		

Table III.5: HotSpot JVM Flags

BlockLayoutByFrequency	LoopUnswitching	UseLoopPredicate
BlockLayoutRotateLoops	PartialPeelLoop	UseSuperWord
RangeCheckElimination	DoEscapeAnalysis	AggressiveOpts
BackgroundCompilation	ReassociateInvariants	SplitIfBlocks
OptmizeStringConcat	UseCondCardMark	Inline
UseCodeCacheFlushing	UseAESIntrinsics	UseAES
EliminateAllocations		

Search Space Construction

The architectural diagram is as follows:

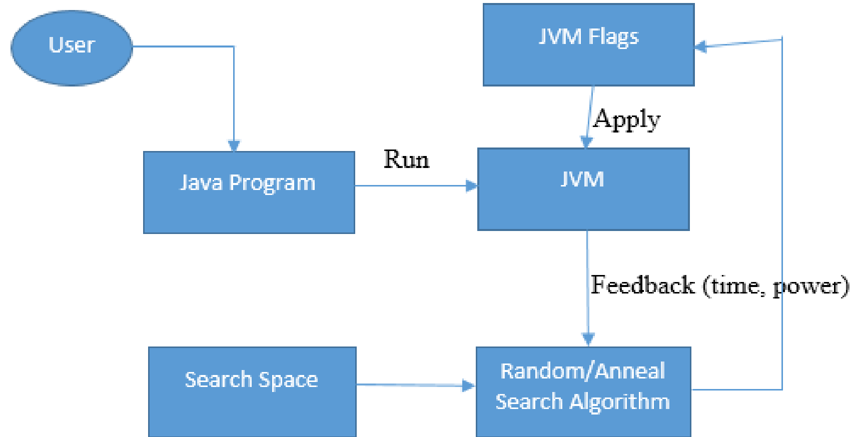


Figure III.2: Architecture Diagram

The search space of combination of JVM parameters is going to be large which will take huge time to compute the total process. Based on the heuristic search, we will reduce some combinations. Here, our algorithm will deal with that machine learning model which can detect negative interaction among optimizations and efface from the search field. After removing the negative

interactions, clustering will be applied and also some other similar techniques.

Heuristic Search

In previous work, it has been shown that choice of the search algorithm has tiny effect on autotuning performance (Qasem and Kennedy, 2006). Here, random search simply samples the JVM flags to a fixed number of times which has been found to be more effective eventually than exhaustive search. For this reason, we decided to implement simulated annealing and random search to generate the output of sequence bit for tuning part.

Top-Most Independent Probability

Here, we have selected k best values and then counted how many times each bit appears in those k sequences. Then we include in the final sequence only those bits that appear more than m times.

Independent Identical probability

We have implemented a statistical technique like IID (Agakov et al., 2006). These techniques can be used to determine the probability of a bit being turned on in a "good" sequence. Here, we have created the final sequence by selecting bits that have a probability $>$ X. PSEAT already has IID implemented in it. The input to this will be a tab delimited file where the first n columns correspond to the n bits and the last column will has the speedup. The number of rows will be the instances searched.

Statistical Modeling

Linear Regression

Linear regression is an approach for modeling the relationship between a dependent variable y and independent variables. Here, the relationships are

modeled using predictor functions where unknown model parameters are estimated from the data. Most commonly, the conditional mean is assumed to be an affine function of independent variables. According to the rule, we can make the output of speedup/powerup is the relation of JVM flags.

Decision Tree

A decision tree is a flowchart-like structure where each node represents a test on an attribute, and each branch represents the outcome of the test and each leaf node represents a class label. The paths from root to leaf represent classification rules. A decision tree and the closely related diagram are used as a visual and analytical decision support tool. Using decision tree, we also make an relationship between JVM flags.

Analysis and Recommendation

As we mentioned earlier, we have used a number of processes to make the analysis and the recommended flags based on that. Most of the HPC applications do not make any user interface. (Burtscher et al., 2010) has developed PerfExpert which is such a tool that combines a simple user interface with a sophisticated analysis engine to identify core, socket and node-level performance bottlenecks in each important procedure. Here, we also have developed a GUI application that allows performance engineers and programmers to inspect and analyze the data in an intuitive manner. Furthermore, this application also provides recommendations about optimizations to the user.

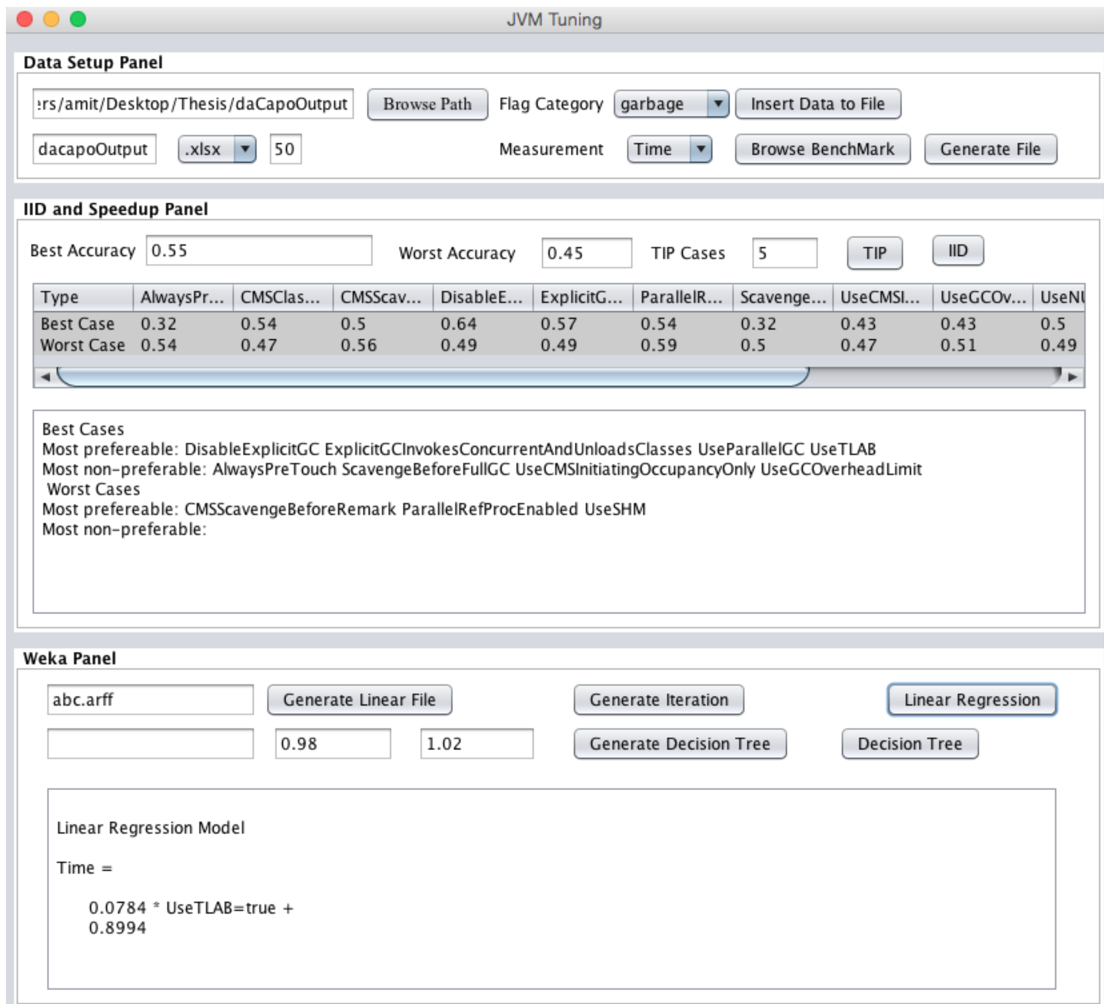


Figure III.3: JVM Tuning GUI

The application is divided into three parts. First part is used to extract data from the jtuner framework and generate the optimize output file. Let's take a look at the 1st part:

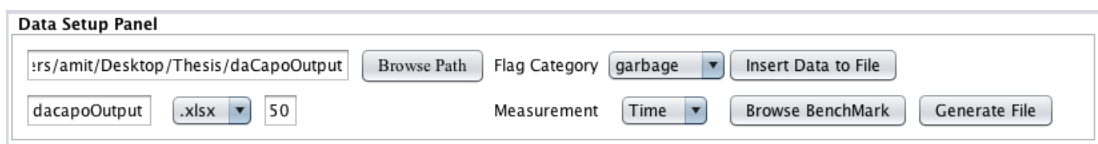


Figure III.4: Data Setup Panel

Firstly, you have to browse the folder path where two folders named "Time" and "Energy" should be there. The output files of the JTuner framework will be stored in the corresponding folder. The output file of the JTuner framework should be named in such a way that it should have workload

name,Time/Energy,number of iteration of tuner,the algorithm name,category of jvm flag. Then, you have to browse the standard excel benchmark file where all the default values of time and power consumption of each workload will be saved. Then we need to input the excel file name in the second textbox and generate the excel file using the default values. Pressing 'insert data to file' button gives us the optimized output of each file of time and energy folder and goes to the corresponding row of the excel file. The file should be displayed as follows:

	A	B	C	D	E	F	G	H
1	Program	Iteration	Search Algc	Flag Categc	Standard Ti	50 custom	50 speedup	Sequence
2	avrora	350	anneal	garbage	56.63	5.7	9.935088	0111010100000
3	avrora	350	anneal	jit	56.63	6.04	9.375828	01011100
4	avrora	350	anneal	jvmflags	56.63	6.25	9.0608	1101000001101110111
5	avrora	350	anneal	combined	56.63	5.82	9.730241	1010010111000001101101000100011111
6	avrora	350	random	garbage	56.63	5.81	9.746988	0100001001010
7	avrora	350	random	jit	56.63	6.16	9.193182	01100101
8	avrora	350	random	jvmflags	56.63	5.85	9.680342	1010101110100100010
9	avrora	350	random	combined	56.63	5.79	9.780656	1111001011010110100110101100011110
10	batik	350	anneal	garbage	3.53	3.1	1.13871	0000010001100
11	batik	350	anneal	jit	3.53	3.04	1.161184	01100010
12	batik	350	anneal	jvmflags	3.53	3.07	1.149837	1101101000110111001
13	batik	350	anneal	combined	3.53	3.26	1.082822	0010001110110011101000001111100101
14	batik	350	random	garbage	3.53	3.07	1.149837	1001000111101
15	batik	350	random	jit	3.53	3.05	1.157377	11001001
16	batik	350	random	jvmflags	3.53	3.01	1.172757	1100010011111100011
17	batik	350	random	combined	3.53	3.05	1.157377	1100001101100001101010110111110111
18	eclipse	350	anneal	garbage	122.73	33.26	3.690018	1101110110110
19	eclipse	350	anneal	jit	122.73	32.85	3.736073	01100001
20	eclipse	350	anneal	jvmflags	122.73	32.77	3.745194	1111010111000100010
21	eclipse	350	anneal	combined	122.73	33.32	3.683373	1001001001110010011010010010011101
22	eclipse	350	random	garbage	122.73	33.06	3.712341	0110111011101
23	eclipse	350	random	jit	122.73	32.22	3.809125	11000101
24	eclipse	350	random	jvmflags	122.73	32.28	3.802045	0001010010000110010
25	eclipse	350	random	combined	122.73	32.84	3.737211	1000100111111000110100111100111101
26	fop	350	anneal	garbage	0.2	0.14	1.428571	0101100001011
27	fop	350	anneal	jit	0.2	0.14	1.428571	11001101
28	fop	350	anneal	jvmflags	0.2	0.11	1.818182	0010110000001111010
29	fop	350	anneal	combined	0.2	2.74	0.072993	1110101110101011101101111110010100
30	fop	350	random	garbage	0.2	2.71	0.073801	0010100001100
31	fop	350	random	jit	0.2	2.57	0.077821	01100000
32	fop	350	random	jvmflags	0.2	2.54	0.07874	1110011110000100001
33	fop	350	random	combined	0.2	2.55	0.078431	0111111100111010110000111010111100
34	h2	350	anneal	garbage	96.67	100.62	0.960743	0001111100111
35	h2	350	anneal	jit	96.67	104.24	0.927379	11110001
36	h2	350	anneal	jvmflags	96.67	104.93	0.921281	1110110100101111100
37	h2	350	anneal	combined	96.67			
38	h2	350	random	garbage	96.67	16.89	5.723505	0011010101111
39	h2	350	random	jit	96.67	17.16	5.63345	11011110

Figure III.5: Sample Excel File

Now in the second part, we have to deal two algorithms named Top-Most Independent Probability (TIP) and Independent Identical Probability (IID).

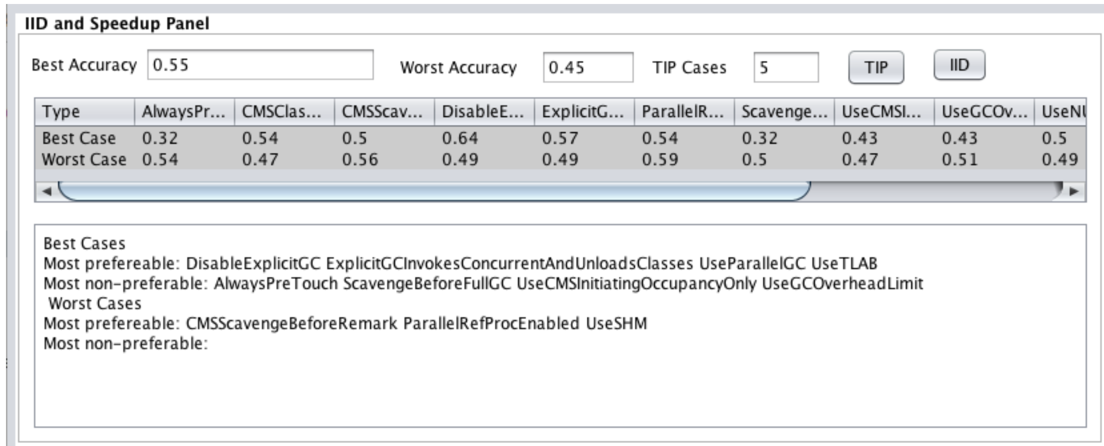


Figure III.6: Algorithm Panel

Here, the input fields are the two probabilities that will be considered as good and bad accuracy and also the number of cases for TIP algorithm. Pressing 'TIP' button will return the recommended flags for best and worst cases based on the previous dataset panel measurement and flag type. And pressing 'IID' button will also return the same stuffs.

In the third panel, the diagram is as follows:

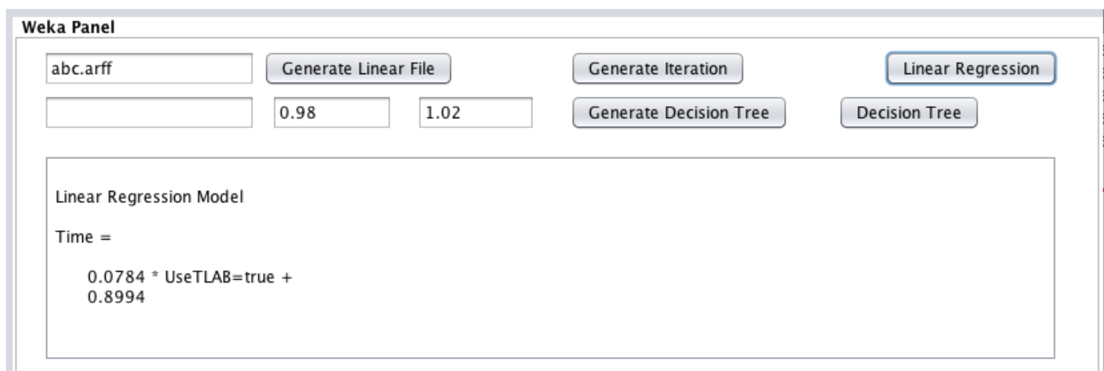


Figure III.7: Algorithm Panel

Here, we will get outputs for two statistical models named as Linear Regression and Decision Tree using weka (Holmes et al., 1994). Before getting the linear regression value, we have to generate linear regression weka file from the selected folder based on the selected measurement and flag type. After generating weka file(.arff extension), we will get the final linear regression output from the .arff file. The same approach will also be applicable for decision tree generation.

IV. EXPERIMENTAL RESULTS

In order to measure the validity of the design qualitatively and quantitatively, it has to be tested with several experiments. This chapter explains the environments and setup of the testing design. It starts with a description of a system to run and to test the program. The reason behind all these tuning is to obtain better speedup and powerup. And the equations for these variables are as follows:

speedup = time using customized flags / time with default configuration of flags

powerup = power using customized flags / power with default configuration of flags

Environment and Setup

The prototype system is developed using the GNU Compiler Collection and shell script under linux server. This system has 32,64 bit configuration support at 2.4GHz with 32K L1 cache. Besides, the JTuner application is implemented on Java SDK 8.

Basic Matrix Computation analysis

Initially we have run our jtuner framework on basic matrix operations which are matrix add,multiply,transpose and vector multiply. To run these operations on our framework, we first need to create a makefile and `test.conf` file. In the makefile file, we need to write the command which measures time/energy. The command is like as follows:

```
get_primary.sh -m time -- java (TUNEFLAGS) -cp . MatrixAdd > perf.pseat
```

The sample test.conf file is as follows:

ONLINE

MIN

MFLOPS

350

The above command will give us the computation time of MatrixAdd operation command. We can also get the power value using pwr in lieu of time. The output will be stored in `perf.pseat` file. `TUNEFLAGS` will return the different combination of flags from the framework using random search or simulated annealing algorithm. In the `test.conf` file, the number of iterations will be stored. Here, the number is 350. We also have got all the results for other algorithms such as matrix multiply,transpose and vector multiply like the same ways.

Benchmark Configuration

After the initial small test is done, we have gone through our framework in large scale such as java benchmarks. For testing purpose, we have chosen SPECjvm and DaCapo benchmarks.

Table IV.1: SPECjvm Benchmarks

startup.helloworld	startup.crypto.signverify	startup.scimark.sparse
startup.compiler.compiler	startup.mpegaudio	startup.serial
startup.compiler.sunflow	startup.scimark.fft	startup.sunflow
startup.compress	startup.scimark.lu	startup.xml.transform
startup.crypto.aes	startup.scimark.montecarlo	startup.xml.validation
startup.crypto.rsa	startup.scimark.sor	compiler.compiler

We have run our jtuner framework on SPECjvm benchmark where we have taken 18 workloads. To run these operations on our framework, we first need to create a makefile and test.conf file. In the makefile file, we need to write the

command which measures time/energy. The command is like as follows:

```
get_primary.sh -m time -- java (TUNEFLAGS) -cp . compiler.compiler > perf.pseat
```

The sample `test.conf` file is as follows:

ONLINE

MIN

MFLOPS

350

The above command will give us the computation time of startup.compiler.compiler workload command. We can also get the power value using pwr in lieu of time. The output will be stored in `perf.pseat` file. `TUNEFLAGS` will return the different combination of flags from the framework using random search or simulated annealing algorithm. In the `test.conf` file, the number of iterations will be stored. Here, the number is 350. We also have got all the results for other workloads like the previous way.

We have run our jtuner framework on DaCapo benchmark where we have taken all 14 workloads.

Table IV.2: DaCapo Benchmarks

avrora	batik	eclipse	fop	h2
jython	luindex	lusearch	pmd	sunflow
tomcat	tradebeans	tradesoap	xalan	

To run these operations on our framework, we first need to create a `makefile` and `test.conf` file. In the `makefile` file, we need to write the command which measures time/energy. The command is like as follows:

```
get_primary.sh -m time -- java (TUNEFLAGS) -cp . avrora > perf.pseat
```

The sample `test.conf` file is as follows:

ONLINE

MIN

MFLOPS

350

The above command will give us the computation time of `avrora` workload command. We can also get the power value using pwr in lieu of time. The output will be stored in `perf.pseat` file. `TUNEFLAGS` will return the different combination of flags from the framework using random search or simulated annealing algorithm. In the test.conf file, the number of iterations will be stored. Here, the number is 350. We also have got all the results for other workloads like the previous way.

Lenskit Configuration

Besides java benchmarks, we have also made some computations with recommendation tool Lenskit. For the testing purpose, we have used 3 different Lenskit workloads. They are : `sweepFunkSVDML1M`,`sweepItemItemML1M` and `evaluateCommonML1M`. `sweepFunkSVDML1M` task generates result for funkSVD algorithm (Koren, 2008), `sweepItemItemML1M` for item-item collaborative filter algorithm (?). Finally, `evaluateCommonML1M` produces result for the combination of the algorithms including personalized mean, user-user collaborative filtering (Sneha and Varma, 2015), item-item collaborative filtering (?) and funkSVD (Koren, 2008) algorithm. Here, all the tasks deal with 1M movielens (Lam and Herlocker, 2012) data.

To run these operations on our framework, we first need to create a `makefile` and `test.conf` file in the lenskit folder. In the makefile file, we need to write the command which measures time/energy.

The command is like as follows:

```
get_primary.sh -m pwr -- ./gradlew evaluateCommonML1M -PuseEvalCache=false -Ple
```

The sample `test.conf` file is as follows:

ONLINE

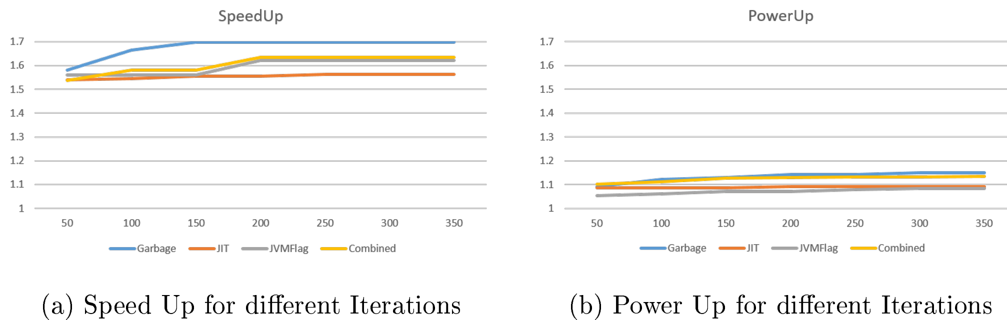
MIN

MFLOPS

350

The above command will give us the computation power of evaluateCommonML1M task command. We can also get the time value using time in lieu of pwr. We also need to pass '-PuseEvalCache=false' to Gradle when you run it, to disable LensKit's evaluator cache. The output will be stored in `perf.pseat` file. `TUNEFLAGS` will return the different combination of flags from the framework using random search or simulated annealing algorithm. In the test.conf file, the number of iterations will be stored. Here, the number is 350. We also have got all the results for other workloads like the previous way.

Test Result of Matrix Computation



(a) Speed Up for different Iterations

(b) Power Up for different Iterations

Figure IV.1: Matrix Analysis

In a nutshell, both the speedup and powerup performs better than the standard value. Here, X-axis represents the number of iteration for tuning and Y-axis represents the average speedup/powerup of each JVM flag category. If we take a deep look at the both the graphs, we have found that advanced garbage flags perform better for both speedup and powerup. And combined flags also perform overall which means that all the flags can also give us a good performance besides with the individual flag categories.

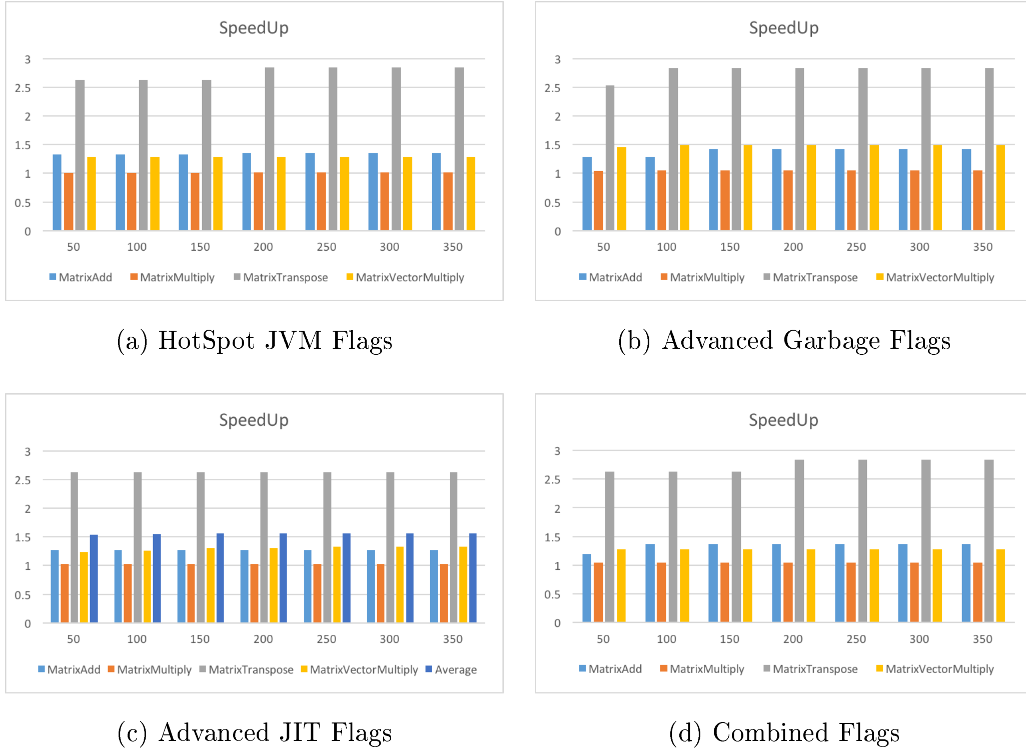


Figure IV.2: Time Analysis For Matrix Tasks

The above figure represents the speedup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each matrix operation. Here, all the speedup result is greater than 1.0 which means all the matrix operations perform better than the default configuration. And after 150 iterations, the optimized value for each flag category is quite steady.

(a) HotSpot JVM Flags

(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.3: Power Analysis For Matrix Tasks

The above figure represents the powerup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each matrix operation. Here, all the speedup result is greater than 1.0 which means all the matrix operations perform better than the default configuration. And after 150 iterations, the optimized value for each flag category is quite steady.

Advanced Garbage flag:

Time:

TIP:

Best Case:

CMSClassUnloadingEnabled	UseNUMA
CMSScavengeBeforeRemark	UseSHM
DisableExplicitGC	

Worst Case:

CMSClassUnloadingEnabled	ScavengeBeforeFullGC
CMSScavengeBeforeRemark	UseCMSInitiaitingOccupancyOnly

IID:

Best Case:

UseSHM

Worst Case:

AlwaysPreTouch	UseNUMA
DisableExplicitGC	UseParallelGC

Linear Regression:

Time=0.039 * UseCMSInitiatingOccupancyOnly + 0.0538 * UseSHM + 0.0534 * UseTLAB + 1.264

Power:

TIP:

Best Case:

AlwaysPreTouch	DisableExplicitGC
CMSClassUnloadingEnabled	UseTLAB
UseGCOverheadLimit	UseParallelGC

Worst Case:

AlwaysPreTouch	UseNUMA
UseGCOverheadLimit	UseSHM
UseCMSInitiatingOccupancyOnly	

Linear Regression:

Power = 0.0268 * ScavengeBeforeFullGC=false +0.0262 * UseParallelGC=true +0.9576

B. Advanced JIT flags:

Time:

TIP:

Best Case:

AggressiveOpts	Inline
DoEscapeAnalysis	UseAES
BackgroundCompilation	

IID:

Best Case:

BackgroundCompilation	Inline

Linear Regression:

Time=0.0346 * BackgroundCompilation + 0.0999 * Inline

Power:

TIP:

Best Case:

UseAES	Inline

Worst Case:

DoEscapeAnalysis	UseAES

Linear Regression:

Power =0.0353 * AggressiveOpts=false +0.0236 * BackgroundCompilation=false +0.9381

C. HotSpot JVM Flags:

Power:

TIP:

Best Case:

DoEscapeAnalysis	SplitIfBlocks
OptimizeStringConcat	UseAESIntrinsics
UseCodeCacheFlushing	

Worst Case:

BlockLayoutRotateLoops	LoopUnswitching
RangeCheckElimination	SplitIfBlocks
OptimizeStringConcat	UseAES
UseCodeCacheFlushing	

IID:

Worst Case:

BackgroundCompilation

Linear Regression:

Power =0.0208 * Inline =true +0.0222 * LoopUnswitching =false +0.0202 * SplitIfBlocks =true +0.0335 * BackgroundCompilation =false + 0.0217 * OptimizeStringConcat =false +0.0244 * UseAES =false +0.8936

Time:

TIP:

Best Case:

BlockLayoutRotateLoops	Inline
LoopUnswitching	PartialPeelLoop
BackgroundCompilation	SplitIfBlocks
OptimizeStringConcat	UseCodeCacheFlushing

Worst Case:

RangeCheckElimination	Inline
ReassociateInvariants	UseCondCardMark
BackgroundCompilation	

IID:

Best Case:

AggressiveOpts	Inline

Linear Regression:

Time=0.1092 * Inline + 0.0408 * SplitIfBlocks + 0.0605 * BackgoundCompilation + 0.0619 * UseAESIntrinsics + 1.1058

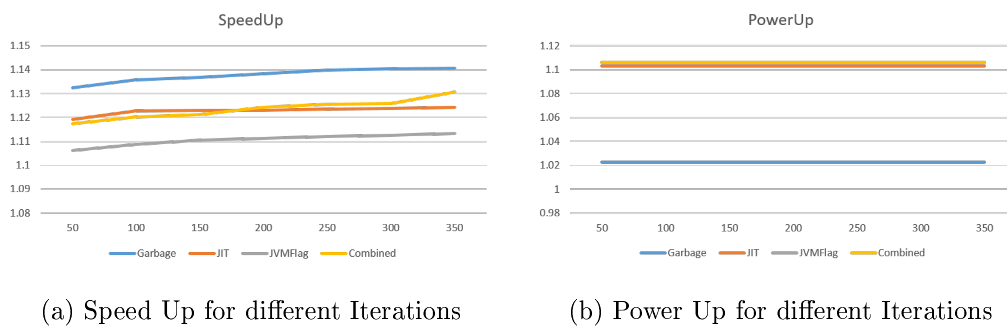
Combined Flags:

Time = 0.0487 * AlwaysPreTouch=false + 0.0411 * CMSScavengeBeforeRemark=false + 0.0348 * ScavengeBeforeFullGC=true +

0.0518 * UseCMSInitiatingOccupancyOnly=false +0.0491 * UseParallelGC=true +0.0551 * UseAES=false +0.0357 * UseCondCardMark=false +0.0603 * UnlockDiagnosticVMOptions=false + 0.3992

Power =0.0289 * AlwaysPreTouch=true +0.0194 * ExplicitGCInvokesConcurrentAndUnloadsClasses=false +0.019 * ParallelRefProcEnabled=true +0.0238 * UseCMSInitiatingOccupancyOnly=true +0.0232 * Inline=true +0.0185 * ReassociateInvariants=false +0.0214 * BackgroundCompilation=false +0.8961

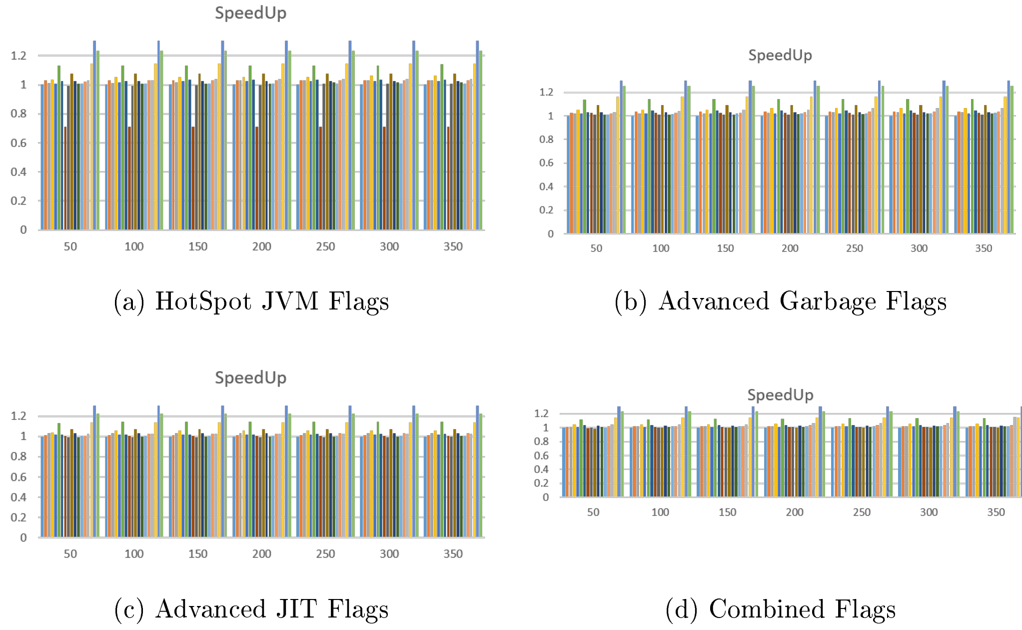
Test Result of SPECjvm Benchmark



(a) Speed Up for different Iterations (b) Power Up for different Iterations

Figure IV.4: SPECjvm Analysis

In a nutshell, both the speedup and powerup performs better than the result without tuning. Here, X-axis represents the number of iteration for tuning and Y-axis represents the average speedup/powerup of each JVM flag category. If we take a deep look at the both the graphs, we have found that advanced garbage flags perform relatively best in speedup case but performs slightly bad than other categories for powerup. And combined flags also perform overall which means that all the flags can also give us a good performance besides with the individual flag categories.



(a) HotSpot JVM Flags

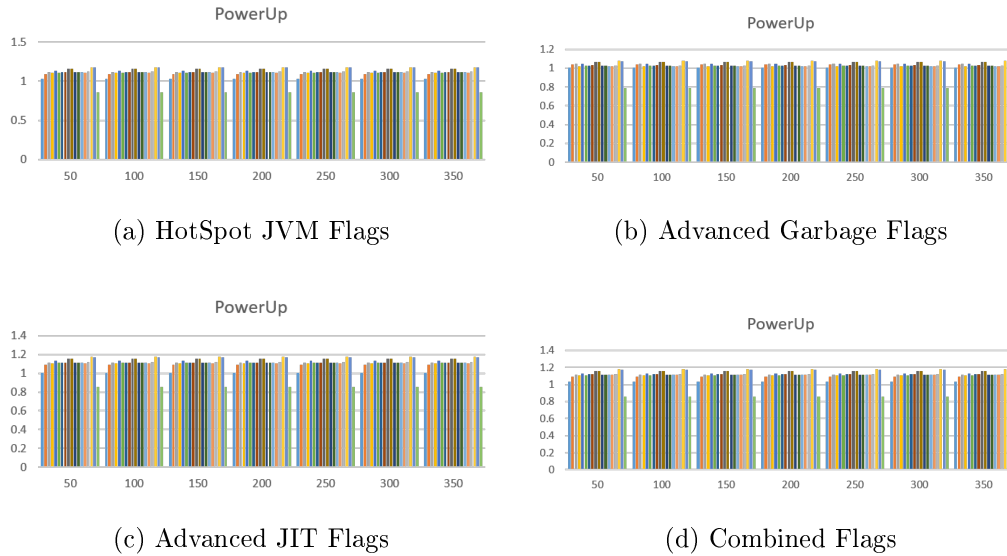
(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.5: Time Analysis For SPECjvm Tasks

The above figure represents the speedup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, all the speedup result is greater than 1.0 except startup.helloworld workload which means all the SPECjvm workloads perform better than the default configuration. Actually, startup.xml.transform workload speedup is almost double for all cases.



(a) HotSpot JVM Flags

(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.6: Power Analysis For SPECjvm Tasks

The above figure represents the powerup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, all the powerup result is greater than 1.0 except startup.helloworld workload which means all the SPECjvm workloads perform better than the default configuration. And after 150 iterations, the optimized value for each flag category is quite steady.

Advanced Garbage flag: Time:

TIP:

Best Case:

CMSClassUnloadingEnabled	UseNUMA	UseParallelGC	UseTLAB

IID: Best Case:

UseParallelGC	UseTLAB

Worst Case:

UseGCOverheadLimit	UseSHM

Linear Regression:

Time=0.3966 * CMSClassUnloadingEnabled=true +0.4218 *

DisableExplicitGC=false +0.4905 * UseParallelGC=true +1.4635

Power:

TIP:

Best Case:

UseParallelGC	CMSScavengeBeforeRemark
DisableExplicitGC	UseSHM
ExplicitGCInvokesConcurrentAndUnloadsClasses	

IID:

Best Case:

UseParallelGC	CMSClassUnloadingEnabled
ScavengeBeforeFullGC	UseSHM
ExplicitGCInvokesConcurrentAndUnloadsClasses	

Worst Case:

UseCMSInitiatingOccupancyOnly

Linear Regression:

Power = 0.0246 * ExplicitGCInvokesConcurrentAndUnloadsClasses=true

+1.0244

B. Advanced JIT flags:

Time:

TIP:

Best Case:

UseSuperWord	Inline
BackgroundCompilation	

IID:

Best Case:

BackgroundCompilation	UseCondCardMark

Worst Case:

UseCodeCacheFlushing

Power:

TIP:

Best Case:

UseCondCardMark	AggressiveOpts
DoEscapeAnalysis	Inline

Worst Case:

UseCondCardMark

Linear Regression:

Power = 0.0803 * BackgroundCompilation=false +0.0229 *

DoEscapeAnalysis=false +0.0226 * Inline=true +1.0162

C. HotSpot JVM Flags:

Time:

TIP:

Best Case:

BackgroundCompilation	UseSuperWord
BlockLayoutRotateLoops	Inline
UseLoopPredicate	EliminateAllocations

IID:

Best Case:

BackgroundCompilation	DoEscapeAnalysis
BlockLayoutRotateLoops	EliminateAllocations

Worst Case:

BlockLayoutRotateLoops

Power:

TIP:

Best Case:

OptimizeStringConcat	UseCodeCacheFlushing
DoEscapeAnalysis	AggressiveOpts
RangeCheckElimination	Inline

IID:

Best Case:

BackgroundCompilation	BlockLayoutRotateLoops
DoEscapeAnalysis	EliminateAllocations
OptimizeStringConcat	UseSuperWord
UseCondCardMark	LoopUnswitching

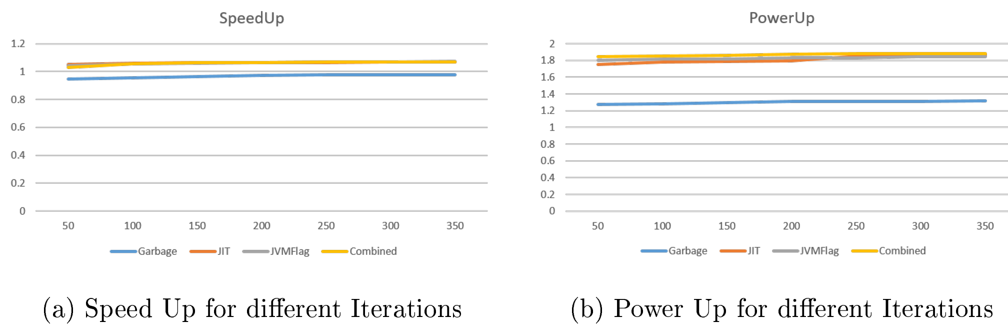
Worst Case:

AggressiveOpts	OptimizeStringConcat
BlockLayoutRotateLoops	UseAES
UseAESInstrinsics	ReassociateInvariants

Linear Regression:

Time=0.404 * SplitIfBlocks =false +0.3808 * UseCondCardMark=true +1.6094

Test Result of DaCapo Benchmark



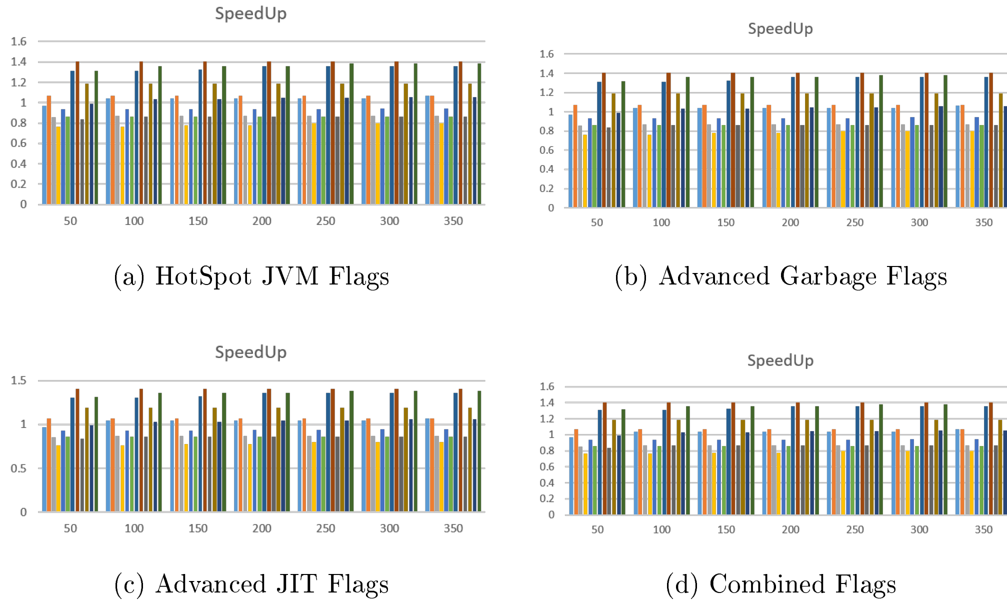
(a) Speed Up for different Iterations

(b) Power Up for different Iterations

Figure IV.7: DaCapo Analysis

In a nutshell, both the speedup and powerup performs better than the result without tuning. Here, X-axis represents the number of iteration for tuning and Y-axis represents the average speedup/powerup of each JVM flag category. If we take a deep look at the both the graphs, we have found that advanced garbage flags perform relatively best in speedup case but performs slightly bad than

other categories for powerup. And combined flags also perform overall which means that all the flags can also give us a good performance besides with the individual flag categories.



(a) HotSpot JVM Flags

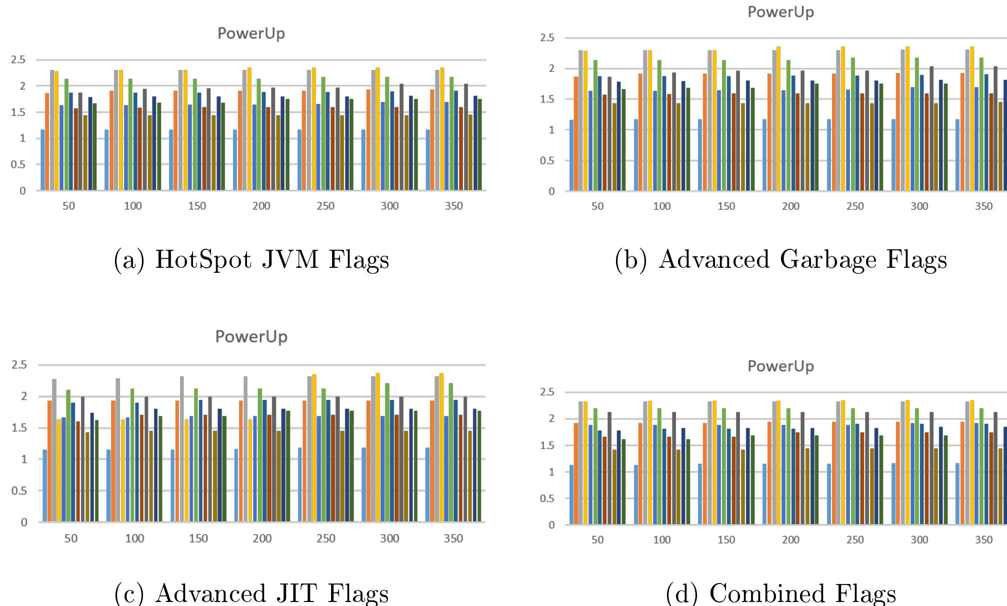
(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.8: Time Analysis For DaCapo Tasks

The above figure represents the speedup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, we have discarded the speedup for tradebean and tradesoap as these never show better result than default one. On the other hand, luindex,sunflow and tomcat workloads perform around 20-40% better than our benchmark value.



(a) HotSpot JVM Flags

(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.9: Power Analysis For DaCapo Tasks

The above figure represents the powerup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, we have discarded the powerup for tradebean and tradesoap as these never show better result than default one. On the other hand, batik, eclipse, h2, jython, luindex and sunflow workloads perform around 40-50% better than our benchmark value.

A. Advance Garbage Flags:

TIP

Best Case:

ExplicitGCInvokesConcurrentAndUnloadsClasses	UseSHM
CMSScavengeBeforeRemark	UseParallelGC

Worst Case:

IID:

Best Case:

UseParallelGC	UseTLAB

Worst Case:

DisableExplicitGC

Linear Regression

Time=0.0784 * UseTLAB=true +0.8994

Power

TIP:

Best Case:

UseTLAB	
UseGCOverheadLimit	UseNUMA
ExplicitGCInvokesConcurrentAndUnloadsClasses	

Worst Case:

CMSScavengeBeforeRemark	DisableExplicitGC

IID:

Worst Case:

CMSScavengeBeforeRemark

Linear Regression:

Power = 0.014 * ExplicitGCInvokesConcurrentAndUnloadsClasses=false + 0.0116 * ScavengeBeforeFullGC=false +0.0112 * UseNUMA=false +0.0385 * UseParallelGC=false +1.0767

B. Advanced JIT flags:

TIP

Best Case:

UseCodeCacheFlushing	Inline
BackgroundCompilation	

Worst Case:

UseSuperWord

IID

Best Case:

BackgroundCompilation

Linear Regression:

Time=0.0749 * AggressiveOpts=false +0.2739 * BackgroundCompilation=true +0.1367 * Inline=false +0.6505

Power:

TIP

Best Case:

Inline

IID:

Worst Case:

BackgroundCompilation

Linear Regression:

Power =0.3889 * BackgroundCompilation=false +0.0232 * Inline=false +1.1502

C. Hotspot JVM flags

Time

TIP:

Best Case:

UseCodeCacheFlushing	Inline
DoEscapoAnalysis	OptimizeStringConcat
BackgroundCompilation	UseAESIntrinsics

IID:

Best Case:

BackgroundCompilation	UseCondCardMark

Linear Regression:

Time=0.0671 * DoEscapeAnalysis =false +0.0866 * EliminateAllocations =true +0.0856 * Inline =false +0.0769 * PartialPeelLoop =true +0.0833 * UseLoopPredicate =false +0.0987 * UseSuperWord =false +0.2354 * BackgroundCompilation =true +0.522

Power

TIP:

Best Case:

BlockLayoutByFrequency	ReassociateInvariants
DoEscapoAnalysis	UseAESIntrinsics
UseCondCardMark	

Worst Case:

SplitIfBlocks

IID:

Best Case:

BlockLayoutByFrequency

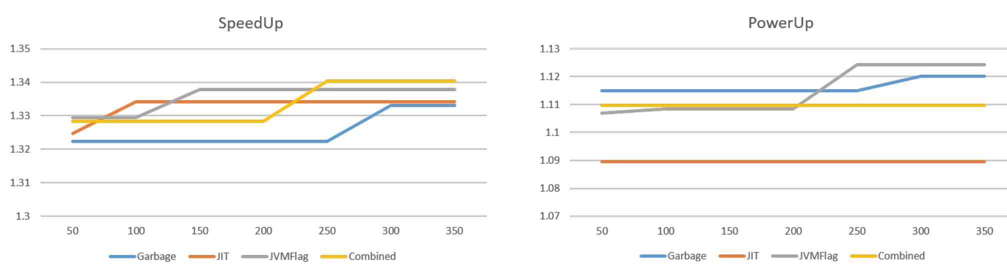
Worst Case:

BackgroundCompilation

Linear Regression:

Power =0.0122 * LoopUnswitching =true +0.025 * PartialPeelLoop =false +0.0184 * AggressiveOpts =false +0.4122 * BackgroundCompilation =false +0.017 * UseAES =false +0.0121 * UseCondCardMark=false +1.1298

Test Result of Lenskit tool



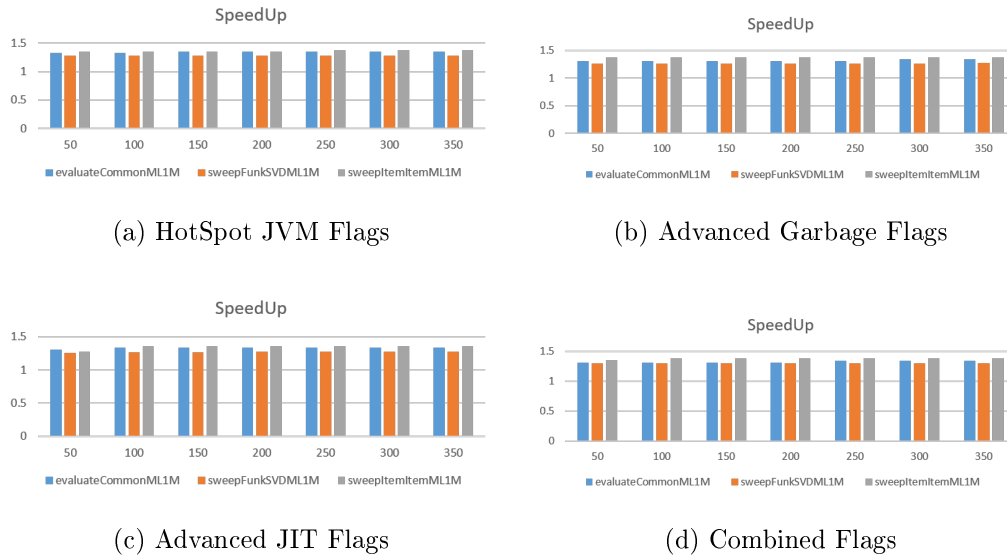
(a) Speed Up for different Iterations

(b) Power Up for different Iterations

Figure IV.10: Lenskit Analysis

In a nutshell, both the speedup and powerup performs better than the result without tuning. Here, X-axis represents the number of iteration for tuning and Y-axis represents the average speedup/powerup of each JVM flag category. If we take a deep look at the both the graphs, we have found that combined flags

perform relatively best in speedup case but jvm flags performs better for powerup. And combined flags also perform overall which means that all the flags can also give us a good performance besides with the individual flag categories.



(a) HotSpot JVM Flags

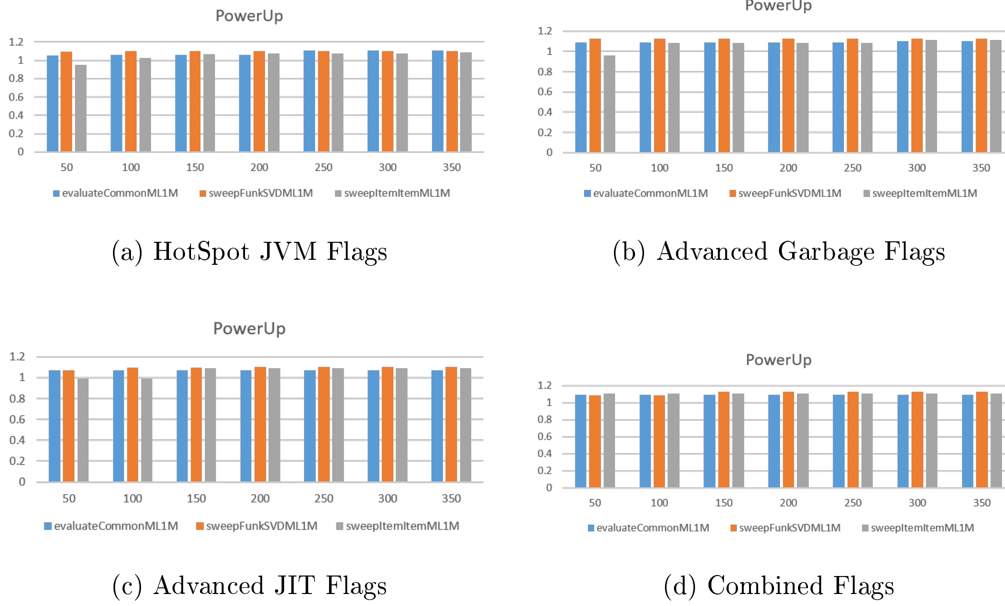
(b) Advanced Garbage Flags

(c) Advanced JIT Flags

(d) Combined Flags

Figure IV.11: Time Analysis For Lenskit Tasks

The above figure represents the speedup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, all three workloads perform around 25-35% better than our benchmark value.



(a) HotSpot JVM Flags (b) Advanced Garbage Flags

(c) Advanced JIT Flags (d) Combined Flags

Figure IV.12: Power Analysis For Lenskit Tasks

The above figure represents the powerup of the proposed flag categories for different iteration. Here, X-axis represents the number of iteration that we have used in our jtuner framework and Y-axis represents the speedup of each workload. Here, all three workloads perform around 5-10% better than our benchmark value.

A. Advance Garbage Flags:

TIP

Best Case:

UseGCOverheadLimit	UseTLAB
CMSClassUnloadingEnabled	CMSScavengeBeforeRemark

Worst Case:

AlwaysPreTouch	UseParallelGC
UseCMSInitiaitingOccupancyOnly	

Linear Regression:

Time=0.0057 * UseTLAB=false +1.1661

Power

TIP:

Best Case:

AlwaysPreTouch	UseTLAB
ParallelRefProcEnabled	UseSHM

Worst Case:

ParallelRefProcEnabled	UseParallelGC
UseGCOverheadLimit	UseSHM
ExplicitGCInvokesConcurrentAndUnloadsClasses	

IID:

Best Case:

UseParallelGC	UseTLAB

B. Advanced JIT flags:

TIP

Best Case:

AggressiveOpts	UseCondCardMark
BackgroundCompilation	UseAES

Worst Case:

AggressiveOpts

Linear Regression:

Time=0.0067 * AggressiveOpts=true +1.173

Power:

TIP

Best Case:

UseCondCardMark

Worst Case:

UseCondCardMark	UseSuperWord	UseAES

IID:

Best Case:

UseCondCardMark

Worst Case:

UseAES

Linear Regression:

Power =0.007 * UseAES=false +0.9687

C. Hotspot JVM flags

Time

TIP:

Best Case:

RangeCheckElimination	DoEscapeAnalysis
BlockLayoutByFrequency	BlockLayoutRotateLoops
ReassociateInvariants	SplitIfBlocks
BackgroundCompilation	OptimizeStringConcat
UseAESInstrinsics	UseAES
UseCodeCacheFlushing	

Worst Case:

BlockLayoutByFrequency	BlockLayoutRotateLoops
EliminateAllocations	ReassociateInvariants

IID:

Worst Case:

BlockLayoutRotateLoops	LoopUnswitching
BackgroundCompilation	PartialPeelLoop
SplitIfBlocks	UseLoopPredicate
UseAESIntrinsics	UseCondCardMark

Power

TIP:

Best Case:

BlockLayoutByFrequency	ReassociateInvariants
LoopUnswitching	PartialPeelLoop
UseCondCardMark	UseSuperWord
BackgroundCompilation	OptimizeStringConcat
UseCodeCacheFlushing	

Worst Case:

BlockLayoutByFrequency	BlockLayoutRotateLoops
DoEscapeAnalysis	ReassociateInvariants
UseLoopPredicate	UseCondCardMark
BackgroundCompilation	

IID:

Best Case:

BlockLayoutByFrequency	PartialPeelLoop
DoEscapeAnalysis	EliminateAllocations
UseAES	UseCondCardMark
BackgroundCompilation	

Worst Case:

UseSuperWord

Linear Regression:

Power =0.0059 * SplitIfBlocks =false +0.0068 * UseSuperWord =false +0.0051 * UseCondCardMark=true +0.968

Algorithms Comparison

After experimenting on all the benchmarks using JVM flags, we also need to check which algorithm works best for a benchmark. Here, The speedup and powerup comparison for matrix operations are as follows:

Measurement	TIP	IIP	Linear Regression
Speedup	1.303234	1.34029	1.26308
Powerup	1.09385	1.038473	1.09125

Comparing with all the cases, linear regression performs best for speedup and TIP for powerup.

Speedup and Powerup comparison of SPECjvm benchmark:

Measurement	TIP	IIP	Linear Regression
Speedup	1.02333	1.025323	1.01123
Powerup	1.00178	1.0013351	1.0006

In this case, we have gained some speedup and powerup but that amount is not upto mark as most of the time it takes default time/energy.

Speedup and Powerup comparison of DaCapo benchmark:

Measurement	TIP	IIP	Linear Regression
Speedup	1.093413	1.10268	1.07068
Powerup	1.019626	1.24489	1.23923

Here, IIP generates the best speedup and powerup result in comparison with other two algorithms.

Speedup and Powerup comparison of Lenskit tool:

Measurement	TIP	IIP	Linear Regression
Speedup	1.177873	1.078317	1.0
Powerup	1.015254	1.012025	1.0

Here, TIP generates the best speedup and powerup result in comparison with other two algorithms.

So, after generating result using algorithm prefered JVM flags, we can say that IIP works best of all other algorithms. Most of the cases, Linear Regression doesn't give us more improvement than default one as the regression coefficient value of the flags are not enough. TIP also works well in both speedup and powerup cases.

Summary

In the experiment section, we have experimented data in different benchmarks. Here, we consider the iteration number as 350 so that we can identify the minimum number of autotuning. After all the experiments, we can say that minimum 150 number of iterations are enough to get the optimized time and power consumption. As we noted earlier, we have experimented data in different type of JVM flags. From the experimental results, we can say that UseParallelGC and UseTLAB of Advance Garbage flag category are good for time consumption and the latter one is also good for power consumption. In case of JIT flags, BackgroundCompilation and Inline are good for time and the former one is good for power consumption.

V. CONCLUSION AND FUTURE WORK

Conclusion

In this thesis a multi-objective auto-tuner framework is demonstrated which reliably recommend the flags for general Java based application. A simple Java based application to detect these flags is also described. The empirical evidence shows that the implemented framework can optimize both time and energy successfully and make the impact.

Future Work

There is a need to develop an accurate model of JVM flags which could help reduce the runtime complexity, and allow the developer to use those flags based on a specific algorithm. We can use other machine learning models mentioned in Weka to get different recommended flags. In our work, the simulated annealing algorithm should be more heuristic in such a way that one flag will be enable depending on its homogeneous flag. Future work will be needed to integrate other algorithm such as direct search for more accurate generated output for JVM flags. Future work will also be needed to obtain a reliable model of other integer type of JVM flags which may be good for more accurate result. A better model could detect the time and power consumption more precisely, resulting in better tracking.

REFERENCES

Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F., Thomson, J., Toussaint, M., and Williams, C. K. (2006). Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 295–305. IEEE Computer Society.

Baker, M., Carpenter, B., and Shaft, A. (2006). Mpj express: towards thread safe java hpc. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE.

Bautista, D., Sahuquillo, J., Hassan, H., Petit, S., and Duato, J. (2008). A simple power-aware scheduling for multicore systems when running real-time applications. In *IPDPS*.

Burtscher, M., Kim, B.-D., Diamond, J., McCalpin, J., Koesterke, L., and Browne, J. (2010). Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society.

Contreras, G. and Martonosi, M. (2005). Power prediction for intel xscale reg; processors using performance monitoring unit events. In *Low Power Electronics and Design, 2005. ISLPED '05. Proc. of the 2005 Int'l Symposium on*, pages 221–226.

Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R., and Schulz, M. (2008). Prediction models for multi-dimensional power-performance optimization on many cores. In *PACT*.

Ding, Y., Ansel, J., Veeramachaneni, K., Shen, X., O'Reilly, U.-M., and Amarasinghe, S. (2015). Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 379–390. ACM.

Ekstrand, M. D., Ludwig, M., Konstan, J. A., and Riedl, J. T. (2011). Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *Proceedings of the fifth ACM conference on Recommender systems*, pages 133–140. ACM.

Fernando, M., Rusira, T., Perera, C., and Philips, C. Cgo: U: Auto-tuning the hotspot jvm.

Fursin, G., Kashnikov, Y., Memon, A. W., Chamski, Z., Temam, O., Namolaru, M., Yom-Tov, E., Mendelson, B., Zaks, A., Courtois, E., et al. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, 39(3):296–327.

Gordon, R. (1998). *Essential JNI: Java Native Interface*. Prentice-Hall, Inc.

Hall, M., Chame, J., Chen, C., Shin, J., Rudy, G., and Khan, M. M. (2009). Loop transformation recipes for code generation and auto-tuning. In *Languages and Compilers for Parallel Computing*, pages 50–64. Springer.

Holmes, G., Donkin, A., and Witten, I. H. (1994). Weka: A machine learning workbench. In *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*, pages 357–361. IEEE.

Huang, H.-P., Jeng, J.-C., and Luo, K.-Y. (2005). Auto-tune system using single-run relay feedback test and model-based controller design. *Journal of Process Control*, 15(6):713–727.

Jantz, M. R. and Kulkarni, P. A. (2013a). Exploring single and multilevel jit compilation policy for modern machines 1. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):22.

Jantz, M. R. and Kulkarni, P. A. (2013b). Performance potential of optimization phase selection during dynamic jit compilation. *ACM SIGPLAN Notices*, 48(7):131–142.

Jayasena, S., Fernando, M., Rusira, T., Perera, C., and Philips, C. (2015). Auto-tuning the java virtual machine. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 1261–1270. IEEE.

Karau, H., Konwinski, A., Wendell, P., and Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. " O'Reilly Media, Inc.".

Kim, K. H., Buyya, R., and Kim, J. (2007). Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *IEEE Symposium on Cluster Computing and the Grid*, CCGRID '07.

Koren, Y. (2008). Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM.

Lam, S. and Herlocker, J. (2012). Movielens 1m dataset.

Liao, S.-w., Hung, T.-H., Nguyen, D., Chou, C., Tu, C., and Zhou, H. (2009). Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 56. ACM.

Merkel, A., Stoess, J., and Bellosa, F. (2010). Resource-conscious scheduling for energy efficiency on multicore processors. In *Proc. of the 5th European conference on Computer systems*, EuroSys '10.

Paleczny, M., Vick, C., and Click, C. (2001). The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association.

Qasem, A. and Kennedy, K. (2006). Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS)*, pages 249–258.

Singer, J., Kovoor, G., Brown, G., and Luján, M. (2011). Garbage collection auto-tuning for java mapreduce on multi-cores. *ACM SIGPLAN Notices*, 46(11):109–118.

Sneha, C. and Varma, G. (2015). User-based collaborative-filtering recommendation.

Taboada, G. L., Touriño, J., and Doallo, R. (2008). Java fast sockets: Enabling high-speed java communications on high performance clusters. *Computer Communications*, 31(17):4049–4059.

Taboada, G. L., Touriño, J., and Doallo, R. (2012). F-mpj: scalable java message-passing communications on parallel systems. *The Journal of Supercomputing*, 60(1):117–140.

Teodorescu, R. and Torrellas, J. (2008). Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of the 35th Annual Int'l Symposium on Computer Architecture*, ISCA '08.

Vega, A., Bose, P., and Buyuktosunoglu, A. (2012). Power-aware thread placement in smt/cmp architectures. In *Proc. of the Fourth Workshop on Energy-Efficient Design*.

White, T. (2012). *Hadoop: The definitive guide*. " O'Reilly Media, Inc.".