CGO:U:Auto-tuning the HotSpot JVM

Milinda Fernando, Tharindu Rusira, Chalitha Perera, Chamara Philips
Department of Computer Science and Engineering
University of Moratuwa
Sri Lanka
{milinda.10, tharindurusira.10, chalitha.10, chcphilips.10}@cse.mrt.ac.lk
Supervisors: Prof. Sanath Jayasena (sanath@cse.mrt.ac.lk), Prof. Saman Amarasinghe (saman@csail.mit.edu)

Abstract— **We address the problem of tuning the performance of the Java Virtual Machine (JVM) with run-time flags (parameters). We use the HotSpot VM in our study. To the best of our knowledge, ours is the first auto-tuner for optimizing the performance of the JVM as a whole. We organize the JVM flags into a tree structure by building a flag-hierarchy, which helps us to resolve dependencies on different aspects of the JVM such as garbage collector algorithms and JIT compilation, and helps to reduce the configuration search-space. Experiments with the SPECjvm2008 and DaCapo benchmarks show that we could optimize the HotSpot VM with significant speedup; 16 SPECjvm2008 startup programs were improved by an average of 19% with three of them improved dramatically by 63%, 51% and 32% within a maximum tuning time of 200 minutes for each. Based on a minimum tuning time of 200 minutes, average performance improvement for 13 DaCapo benchmark programs is 26% with 42% being the maximum improvement.**

I. PROBLEM AND MOTIVATION

In performance tuning, we attempt to select parameters and their values to improve performance of a system. In the case of performance tuning of software on a given hardware/OS platform, source-level optimizations, using compiler features and specialized tools and libraries are common techniques. Performance tuning can be done manually or automatically. Manual tuning can sometimes become infeasible due to the complexity induced by a large number of parameters. Auto-tuning is the process of finding the best parameters automatically in order to optimize the performance of a system.

A key problem with auto-tuners is that they are highly domain specific. To overcome this limitation, a generic tuning framework like OpenTuner [1], a framework for building domain specific multi-objective auto-tuners can be used.

In this paper, we address the problem of tuning the Java Virtual Machine (JVM) using run-time flags to improve the performance of any given Java application. By "flags" we mean all kinds of parameters and options (including Boolean settings whose status can be either ON or OFF) that are supported by the JVM and can be specified at the time the Java runtime environment is started. Depending on the machine class (i.e. client or server) and other factors, the JVM will automatically assign default values for many run-time flags; we refer to this as the "default configuration" in this paper.

We present a methodology to improve the performance of a Java program by tuning JVM flags. We selected the widely used OpenJDK HotSpot VM for our work and developed the "HotSpot Auto-tuner" which is described in this paper. Manually tuning the JVM is infeasible because the HotSpot VM has over 600 flags. Further, manual systems tuning can result in suboptimal performance improvements. Our approach, that uses the OpenTuner framework [1], is proved to be effective with experimental results. In our experiments, SPECjvm2008 startup benchmarks are used to analyse the behaviour of the JVM with regard to program startup performance while DaCapo benchmarks are used to test the steady state performance of the JVM. In our approach, we study the effect of the values taken by JVM flags towards the overall performance of the JVM. To do this, we construct an experimental framework where we measure performance improvement of a Java program against different classes of JVM flags such as GC flags, JIT flags, GC+JIT flags combined, and finally, all the flags, considering the JVM as a whole.

II. BACKGROUND AND RELATED WORK

Program auto-tuning has been of interest mainly due to its importance in high performance computing. Performance optimization in the context of high-end programs that consume a lot of system resources is very important.

Among the previous attempts to improve the performance of Java applications, it has been a common practice to isolate a certain aspect of the JVM and study only its effect on the overall performance. These considered different categories of flags like garbage collection (GC) flags or just-in-time (JIT) compiler flags. [2], [3], [4], [5] are examples of this approach of isolating subsets of JVM flags.

Arnold et al. [6] and Ishizaki et al. [7] have conducted experiments adopting manual JIT tuning as their approach. However, this process is cumbersome and computationally infeasible in a production environment where tuning time is a critical factor.

There have been other prior work that follow different techniques. Dubach et al. [8] introduce a method for optimizing a compiler by choosing random configurations over a set of selected configuration space. Apart from these software related performance improvements, [9] proposes a hardware translation based method for improving Java performance.

However, we identify two major limitations in these auto-tuning systems. Firstly, the potential performance improvement is limited by the fact that only a subset of the tunable flags are tuned and this is not desirable in a practical scenario. Secondly, these tuners are highly coupled to a particular domain that they were originally built for.

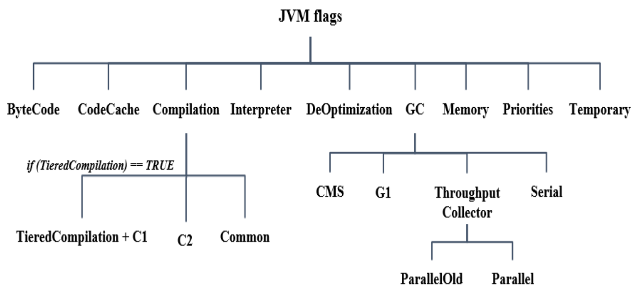

Figure 1. JVM Flag Hierarchy

III. UNIQUENESS OF THE APPROACH

We identify the following features as the unique components of this research.

A. *JVM Flag Hierarchy*

Initial experiments were run with an unstructured set of JVM flags. That is, all JVM flags were fed to the auto-tuner with the same level of significance. This approach however reduces the effectiveness of the tuning process. This is mainly because the auto-tuner is incapable of identifying invalid flag configurations. In this context, we define "invalid configuration" as any flag combination that fails to initialize a JVM instance.

The auto-tuner may search over these invalid configurations making the process inefficient. To overcome the issue, we modelled the dependencies of JVM flags into a hierarchical structure that would enable the HotSpot auto-tuner to prune the theoretical configuration space safely and focus only on the valid configurations.

The flag hierarchy was designed after analyzing HotSpot VM source code (due to lack of documentation) which led us to the realization of how JVM flags are internally related. Later, we show the impact of this improvement on the overall tuning time and the convergence of the HotSpot auto-tuner. Fig. 1 shows the JVM flag hierarchy we developed. No such prior classification of JVM flags were found in the previous literature.

B. *HotSpot Auto-tuner*

We used OpenTuner as a framework to build an auto-tuner for HotSpot VM. Two important components of OpenTuner framework are *configuration manipulator* and the *run function*. *Configuration manipulator* specifies the set of configuration parameters and their possible value ranges, which the HotSpot auto-tuner should search over. The *run function* evaluates the quality of a specific configuration in configuration search space. *Run function, F* maps a configuration to a real number, which is a score representing the quality of the given configuration. (F: C$\rightarrow$ $\mathbb{R}$ where C is the configuration space). Output from the *run function* helps the tuner to compare two different configurations over configuration space and select the better configuration.

As discussed in the previous subsection, we make use of the *JVM Flag Hierarchy* in order to reduce the configuration space that *HotSpot Auto-tuner* would search over. We categorized the related GC flags together. This dramatically reduces the workload given to the HotSpot auto-tuner because once a particular GC is selected, all parameters corresponding to other GC algorithms are

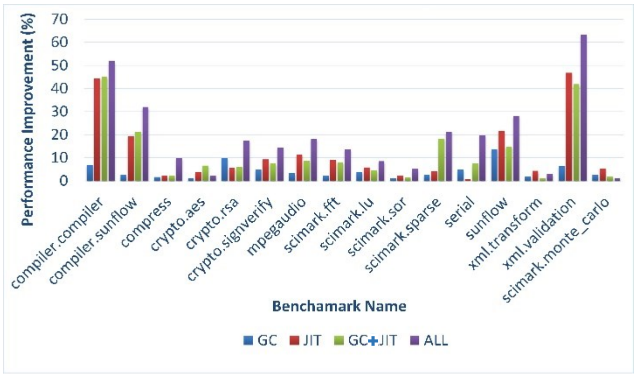

Figure 2. SPECjvm2008 percentage improvements (Architecture 1)

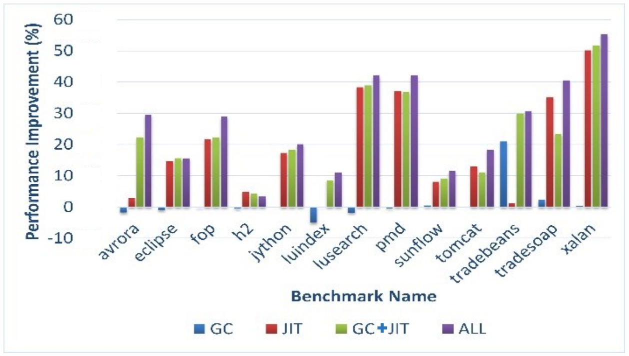

Figure 3. DaCapo percentage improvements (Architecture 1)

TABLE 1. DaCapo improvements on Architecture 2

Program	Default (ms)	Tuned (ms)	% improvement
avrora	6,331	5,581	11.84
eclipse	30,425	22,835	24.95
fop	2,248	1,544	31.32
h2	6,686	5,103	23.67
jython	8,148	8,043	1.29
luindex	1,373	1,184	13.70
lusearch	2,950	1,538	47.86
pmd	3,950	2,310	41.53
sunflow	2,722	2,301	15.44
tomcat	7,176	3,484	51.45
xlan	3,032	1,682	44.54

TABLE 2. DaCapo improvements on Architecture 3

Program	Default (ms)	Tuned (ms)	% improvement
avrora	6,939	6,538	5.78
eclipse	41,215	37,225	9.68
fop	3,613	3,085	14.61
luindex	1,945	1,738	10.62
lusearch	5,814	1,459	74.90
pmd	10,361	5,606	45.89
sunflow	2,901	1,991	31.35
xalan	14,261	3,124	78.10

invalid in the selected context. The same rationale applies for JIT flags as well. The effect of the flag hierarchy is discussed in detail in the next section with respect to the performance improvements of benchmark programs.

The *startup* programs of SPECjvm2008 [10] suite were used to analyze the auto-tuner's effect on the JVM startup time and DaCapo

[11] benchmarks were used to guarantee that achieved performance improvements sustain even after HotSpot VM reaches runtime stability.

C. *JVM Profiling Data*

Runtime statistics of the HotSpot VM are collected using *jstat* profiler under both default and tuned configurations. With these results, we analyze the internal behavioral changes occurred in the Java Virtual Machine as a result of the tuning process. Each benchmark was run with *jstat,* using a profiling interval of 1 ms and the changes in GC, JIT, and class loading behavior are analyzed for DaCapo benchmark programs.

IV. EXPERIMENTAL RESULTS

In this section, we present experimental results obtained by running SPECjvm2008 and DaCapo benchmarks on *HotSpot auto-tuner*. We use following platforms to run experiments.

Architecture 1

Intel Core i7 CPU @3.40Ghz (4 cores), RAM: 16 GB, OS: Ubuntu 12.04 LTS, JVM: OpenJDK 7 (HotSpot VM) update 55

Architecture 2

Intel Core i5-2400 CPU @ 3.10GHz (4 cores), RAM: 4 GB, OS: Ubuntu 14.04 LTS, JVM: OpenJDK 7 (HotSpot VM) update 55

Architecture 3

Intel Xeon E7-4820v2 CPU @2.00 GHz (32 cores), RAM: 64 GB, OS: Ubuntu 14.04 LTS, JVM: OpenJDK 7 (HotSpot VM) update 55

Experiments are run on Architecture 1 to demonstrate the auto-tuning process and other two architectures are used to validate the results and show that the tuning process described here can be extended over different microarchitectures.

In this study, we isolated flags into three different classes, Garbage Collector related flags (GC), Just-In-Time compilation related flags (JIT), and GC & JIT flags combined (GC+JIT). This experimental extension was developed to analyze the tuning behavior of the HotSpot VM with respect to different areas of the JVM such as GC and JIT compilation. These results can be used to demonstrate how different aspects of the JVM affect performance of different programs and how the auto-tuning process changed the internal behavior of the JVM to produce better performance.

Fig.2 and Fig.3 show SPECjvm2008 startup benchmark and DaCapo benchmark results respectively. They show that we could optimize the HotSpot VM with significant speedup; 16 SPEC programs improved with an average of 19.41%, maximum value being 63.37%. Maximum tuning time for SPEC benchmarks was 3.5 hours (210 minutes) for each program. DaCapo programs were tuned with a minimum tuning time of 200 minutes to obtain a tuned configuration. In the first round of experiments, performance of each benchmark program is first measured in their respective performance metrics with the 'default' JVM configuration.

In the second round of experiments, each program was run with a specific, tuned set of JVM flags; the flags and their values for each program were determined by tuning the program with the HotSpot auto-tuner. Values for each performance metric is calculated as the mean value of forty (40) iterations under each configuration with 95% of statistical confidence.

Same experiments were run on the other two architectures with DaCapo benchmarks and the results available as of this writing are shown in Table 1 and Table 2.

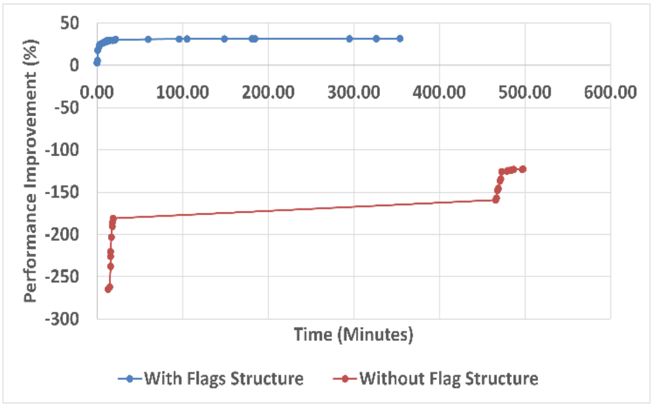

Figure 4. fop tuning with and without the flag hierarchy (Architecture 1)

From our results, it is seen that it is possible to obtain even better performance gains by considering all the tunable flags of the JVM.

We discuss in detail the experimental results under following sub-sections.

A. *Overall Performance Improvement*

The results in Fig 2 and Fig 3 show that, in general, the highest performance improvement for a given benchmark program is obtained when all tunable JVM flags are fed to the HotSpot auto-tuner. Though there are certain exceptions for some programs, the majority of the programs conform to this observation.

As we identified in the study, three most critical components that affect performance of HotSpot VM are garbage collection, JIT compilation and class loading. HotSpot auto-tuner is capable of tuning all three aspects when all the JVM flags are given and the configuration space contains all possible configurations, including the point that generates best performance for a given architecture.

We have proven in this study that the highest achievable JVM performance improvement is obtained by tuning all JVM flags and HotSpot auto-tuner is capable of managing the computational complexity. The potential of the HotSpot auto-tuner as a fast tuner is shown by the fact that most of the benchmark programs achieved their peak performance with a tuning time of 200 minutes.

B. *Flag Hierarchy and Tuning Time*

In most of the offline auto-tuning applications, dedicated tuning time accounts for a considerable overhead. In our experiment, we propose a methodology to reduce this tuning time with respect to the HotSpot VM. The organization of JVM flags into a hierarchical structure helps to prevent the HotSpot auto-tuner from generating invalid or spurious flag configurations. The results 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. Without the flag hierarchy, the tuner takes more time to output the same level of performance improvement for a given benchmark.

Due to the unmanaged computational complexity in the absence of the flag hierarchy, HotSpot auto-tuner tend to test lots of bad

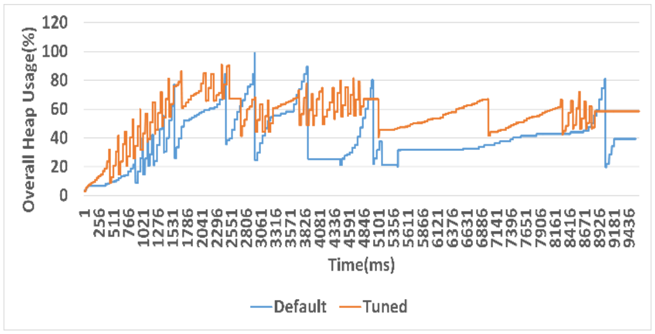

Figure 5. h2 benchmark %HU (Architecture 1)

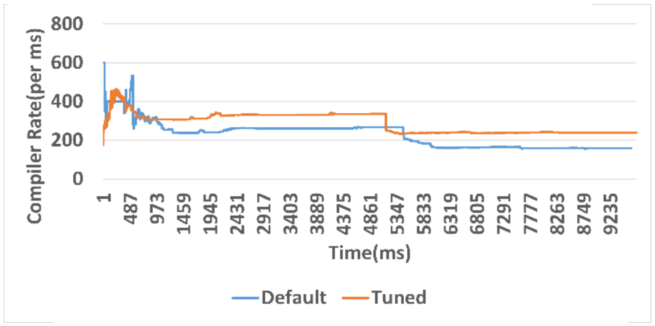

Figure 6. h2 benchmark compilation rate (Architecture 1)

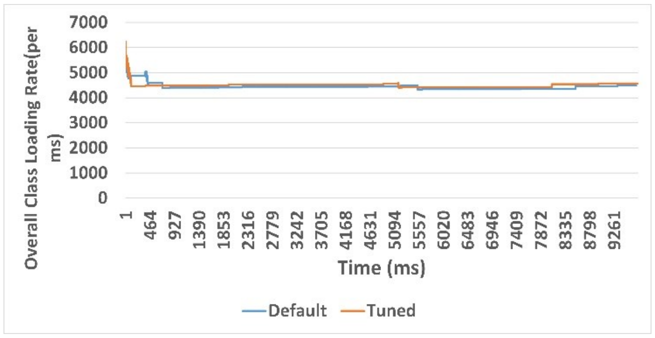

Figure 7. h2 benchmark class load rate (Architecture 1)

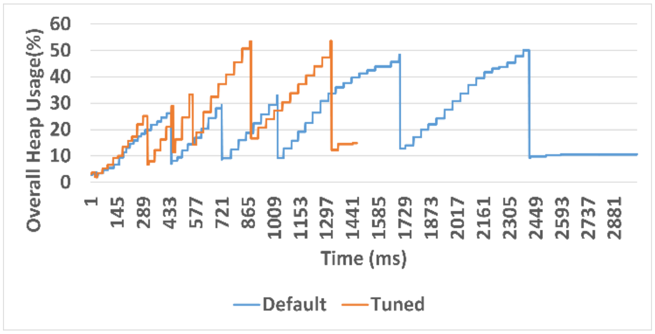

Figure 8. pmd benchmark %HU (Architecture 1)

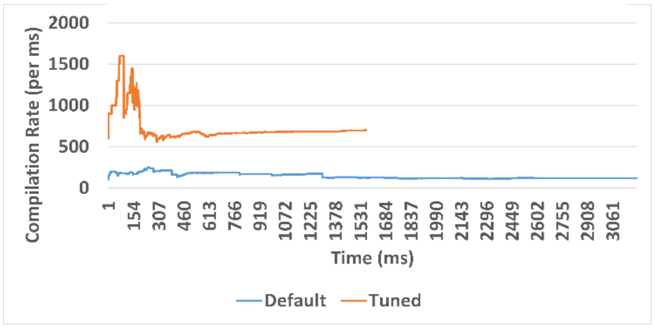

Figure 9. pmd benchmark compilation rate (Architecture 1)

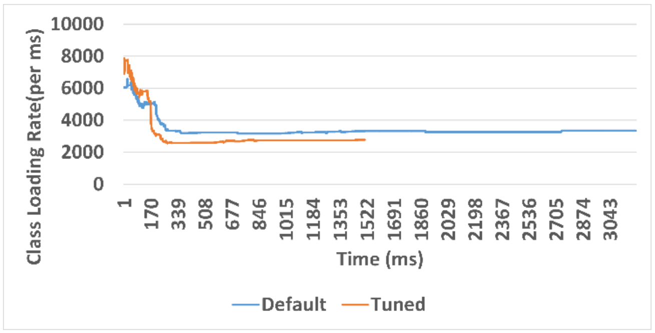

Figure 10. pmd benchmark class loading rate (Architecture 1)

configurations initially, therefore producing performance values below the default performance value. This behavior is common for all the benchmarks and we demonstrate this phenomena using DaCapo fop benchmark. (*See* Fig.4)

C. *Auto-tuning and JVM Internals*

In this section, we study what changes occur in the HotSpot VM as a result of the auto-tuning process. We conduct our analysis based on the results obtained by DaCapo benchmarks as they represent how the HotSpot VM would behave in the steady state. We try to compare our results with *jstat* profiling data in order to explain why the auto-tuning process could improve JVM performance. This analysis is based on recognizable patterns on empirical data.

During our study, we analyze JVM components like Garbage Collection, JIT compilation and JVM class loading process. We picked particular *jstat* metrics in the analysis that provide the overall picture of how GC threads JIT compiler and JVM Class loader behaved under each configuration, default and tuned. Selected JVM measurements are listed below.

- Overall Heap Usage Percentage (%HU)
- Number of Compilation Tasks per millisecond (Compilation Rate)
- Number of Classes Loaded per millisecond (Class Loading Rate)

Percentage Heap Usage can be seen as a measurement of the capability of HotSpot VM to foresee the heap requirement for a given program and allocate necessary heap memory so that the overhead due to the GC threads are reduced. Percentage Heap Usage is calculated using Eq. 1.

$$\%HU_i = \frac{\sum_{k \in \{S0, S1, E, O, P\}} Utilization_k}{\sum_{k \in \{S0, S1, E, O, P\}} Allocation_k} \times 100\% \qquad (1)$$

where $i \in \{\text{default, tuned}\}$. We take summation of the heap utilization and allocation in KB for each heap generations S0 (Survivor space 0), S1 (Survivor space 1), E (Eden space), O (Old space), and P (Permanent space).

The number of compilation tasks is directly taken from *jstat* outputs. The measurements *Compilation Rate* and *Class Loading Rate* are calculated using Eq. 2 where *f(t)* is the number of compilation or class loading tasks performed between time *t-1* and *t*.

$$rate = \frac{\sum_{k=0}^{t} f(k)}{t} \tag{2}$$

The jstat profile data of tuned and default configurations of the 2 selected programs on Architecture 1 are shown in Figs. 5-10 and are discussed below.

1) H2 Benchmark

H2 benchmark executes a number of transactions against a model of a banking application, replacing the HyperSQL database (HSQLDB) benchmark. According to experimental results H2 benchmark depicts relatively low performance improvement compared to other benchmarks.

Considering the %HU shown in Fig.5**,** we can claim that the graph for tuned %HU is more distorted than the default %HU. Tuned %HU shows more rapid fluctuation compared to default configuration which might be caused by unnecessary GC events. Even though tuned %HU contains more variation compared to the default configuration, *Class Loading Rate* of both tuned and default configurations varied in very similar manner (*see* Fig.7). However the *Compilation Rate* (*see* Fig.6) of the H2 benchmark is slightly increased compared to the default configuration. The fluctuated %HU and similar *Class Loading Rate* behavior in Tuned configuration might have decreased the performance while the improvement in the *Compilation Rate* has managed to overcome those decrements and improve the performance of the H2 benchmark in a relatively low manner.

2) PMD Benchmark

PMD benchmark performs an analysis for a set of Java classes for a range of source code problems. As a result of auto-tuning process PMD benchmark has showed a high performance improvement (2nd highest among the DaCapo benchmarks). The %HU and *Class Loading Rate* of the both default and tuned configurations behave in a similar manner (*see* Fig.8, Fig.10). However, the *Compilation Rate* of the Tuned configuration shows a significant improvement (more than 200%) over the default configuration (*see* Fig.9). We can claim that this is a prominent reason for a high performance improvement of the PMD benchmark.

From *jstat* profiling data, optimizations that took place in the JIT compiler are proven to be fundamental for majority of the benchmark programs during their auto-tuning process.

V. CONTRIBUTIONS

In this paper we present how to improve performance of HotSpot VM using *HotSpot auto-tuner* developed using the OpenTuner framework. To prove the tuner's capability, we demonstrated by significantly improving the performance of SPECjvm2008 (startup) and DaCapo benchmark programs after a short tuning period.

The good configurations are not necessarily optimal configurations for a program and are decided based on the initial values of the parameters. This implies that there can be even better configurations and we could potentially achieve them by continuing the tuning process longer or initializing the tuning process with different initial parameter values.

A *JVM Flag Hierarchy* is introduced in this paper that can be used to eliminate invalid combinations of flags and as a result, the auto-tuning process itself is optimized. We show the effect of this hierarchy by auto-tuning programs with and without the support of the flag hierarchy.

JVM runtime is monitored under both default and tuned configurations to gain insight into the internal behavior of the JVM. With a comprehensive analysis of collected profiling data, we show what changes in the JVM are responsible for increased performance of the JVM.

Finally, we identify the significant internal changes occurred in the HotSpot VM as a result of the auto-tuning process. Among them, overall heap usage, overall compilation rate and overall class loading rate are the three most important factors.

VI. REFERENCES

[1] J. Ansel, S. Kamil, K. Veeramachaneni, U.-M. O'Reilly and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in *MIT CSAIL Technical Report MIT-CSAIL-TR-2013-026*, November 1, 2013.

[2] J. Singer, G. Kovoor, G. Brown and M. Luján, "Garbage collection auto-tuning for Java mapreduce on multicores," June 2011.

[3] G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. Irwin and M. Wolczko, "Tuning garbage collection in an embedded Java environment," *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on,* pp. 92-103, 2002.

[4] I. H. Kazi, H. H. Chen, B. Stanley and D. J. Lilja, "Techniques for obtaining high performance in Java programs," *ACM Comput. Surv. 32, 3 (September 2000),* pp. 213-240, 2000.

[5] P. Lengauer and H. Mössenböck, "The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors," in *5th ACM/SPEC international conference on Performance engineering (ICPE '14)*, New York, NY, USA, 2014.

[6] M. Arnold, S. Fink, D. Grove, M. Hind and P. F. Sweeney, "Adaptive optimization in the Jalapeno JVM," *ACM SIGPLAN Notices,* vol. 35, no. 10, pp. 47-65, 2000.

[7] I. Kazuaki, M. Takeuchi, K. Kawachiya, T. Suganuma, O. Gohda, T. Inagaki, A. Koseki, K. Ogata, M. Kawahito, T. Yasue, T. Ogasawara, T. Onodera, H. Komatsu and T. Nakatani, "Effectiveness of cross-platform optimizations for a java just-in-time compiler," in *OOPSLA*, 2003.

[8] Dubach, Christophe, T. M. Jones and M. F. O'Boyle., "Exploring and predicting the architec-ture/optimising compiler co-design space.," in *International conference on Compilers, architectures and synthesis for embedded systems*, 2008.

[9] R. Radhakrishnan, R. Bhargava and L. K. John, "Improving Java performance using hardware translation," in *ICS '01 Proceedings of the 15th international conference on Supercomputing*, New York, NY, USA, 2001.

[10] S. P. E. Corporation, "SPECjvm2008 Benchmarks," [Online]. Available: http://www.spec.org/jvm2008/docs/benchmarks/index.html. [Accessed 5 September 2014].

[11] B. S. M., R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan and e. al, "The DaCapo benchmarks: Java benchmarking development and analysis," *ACM Sigplan Notices,* vol. 41, no. 10, pp. 169-190, 2006.