Investigating the Interaction between Java Programs and Virtual Machines at the Microarchitectural Level

A. Georges L. Eeckhout K. De Bosschere
Department of Electronics and Information Systems (ELIS), Ghent University
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium
{ageorges, leeckhou, kdb}@elis.UGent.be

***Abstract*— In recent years, Java workloads are becoming increasingly prominent on the entire scale of computing devices, ranging from small PDA-like systems to high-end web servers. It becomes thus increasingly important to understand the implications of all the aspects involved when running Java applications when a system is designed. This means that first of all, the interactions between the Java application, its input and the Java Virtual Machine execution the application should be understood. The lowest level on which we can try to comprehend how these components interact is clearly the micro-architectural level. To do this we measure a number of low-level characteristics such as the branch behaviour, the cache behaviour, etc. This is done using the performance counters that can be found on modern microprocessors. Using statistical analysis techniques such as Principal Components Analysis and Cluster Analysis, we seek to gain insights in an understandable way. We conclude that for small input sets, the behaviour is primarily influenced by the virtual machine, while for larger input sets, the Java application itself becomes the primary influence of the low-level behaviour.**

I. Introduction

During the last years, the Java programming language is taking on a more prominent role in the field of software development. Java-based applications can be found on large machines such as web servers, on desktop machines, and finally on small embedded devices such as printers, PDA's and phones. The abundant availability of these applications also introduced a host of virtual machines capable of running them.

We distinguish three aspects of a Java workload that can have a significant influence on the execution behaviour: (i) the virtual machine, (ii) the Java application, and (iii) the input to the application. The virtual machine consists of several closely interacting components, such as the garbage collector, the compiler, the threading system, etc. Because different virtual machines employ different strategies, e.g. mixed-mode interpretation and JIT versus all-out (optimizing) compilation, the choice of JVM can have a large influence on the observed behaviour. Similarly, one can expect a gaming application to behave differently from e.g. a web serving or database application. Finally, large inputs can impose more strain on the memory subsystem of the machine, thus influencing the observed behaviour.

The main question we address is thus the following. How much of the observed behaviour is due to the JVM, the Java application and the input? To find an answer to these questions, we employ the following strategy. First, we gather a large number of execution characteristic measurements, using the performance counters of the observed processor. The resulting data are then analysed using Principal Components Analysis and Cluster Analysis techniques. These techniques aim to present a more understandable view on the gathered data by essentially allowing a reduction in the complexity of the view on the data.

II. Experimental setup

Our experimental setup consists of three important parts, (i) the Java applications, (ii) the virtual machines, and (iii) the hardware platform on which the experiments took place.

We have selected a number of Java applications, coming from the SPECjvm98[1], SPECjbb2000[2] and Java Grande Forum[3] suites. The SPECjvm98 suite contains mostly client-side Java applications, such as a compress program, a compiler, etc. The applications in the suite come with three input set sizes, `s1`, `s10`, and `s100`. The suite was designed to evaluate both hardware aspects (CPU, memory, etc.) as software aspects (virtual machine, kernel activity, etc.) of a Java environment. We have used the `s1` and `s100` input sets. SPECjbb2000 is a server-side benchmark focusing on the middle tier of a three tier (client, business logic, database) system. We have run this benchmark with 2, 4 and 8 warehouses. Finally, the Java Grande Forum benchmark suite is intended to study Java platform performance with so called Grande applications, that require large amounts of memory and processing power. We have used four benchmarks (Euler, Search, MolDyn, and Raytracer) from this suite, with both provided problem sizes.

To run the benchmark applications, we used seven Java virtual machine configurations, i.e. SUN 1.4.1, Blackdown 1.4.1, IBM 1.4.1, JikesRVM 2.2.0 base and adaptive, JRockit 7.1 and Kaffe 1.0.7. Both the SUN and Blackdown machines use the SUN HotSpot virtual machine core [Sun02], which

[1] http://www.spec.org/jvm98
[2] http://www.spec.org/jbb2000
[3] http://www.javagrande.org

employs a mixed mode (interpretation and JIT compilation) scheme, with a generational copying garbage collector. The IBM virtual machine also uses a mixed mode strategy, alternating between IBM's mixed mode interpreter and the IBM JIT compiler [SOT+00]. Kaffe uses interpretation as well as JIT compilation. The remaining two machines, i.e. JRockit [BEA02] and the JikesRVM [AAA+00] configurations, use an all-out compilation scheme. This means that every method is immediately compiled to native code. JRockit and JikesRVM in adaptive mode gather runtime statistics to steer further optimized compilation of hot methods, whereas the JikesRVM base configuration does not perform extra optimizations on the native code it compiled the first time round. JikesRVM has been configured with a non-generational garbage collector, while JRockit employs a generational copying scheme.

The hardware platform we selected is an AMD K7 Duron (model 7) microprocessor [Adv02], running at 1GHz. The Duron is identical to the classic Athlon, but has a smaller L2 cache (64KiB instead of 256KiB). The processor has a 64KiB predecoded L1 instruction cache, and a 64KiB L1 data cache. The L1 fully associative TLB's are also separate for instructions and data, with 24 entries for the L1 I-TLB, and 32 entries for the L1 D-TLB. The L2 four-way set-associative TLB's are both the same size for instructions and data, containing 256 entries each. The processor features a gshare global branch predictor, using 2-bit counters, a 2048-entry branch target buffer, and a 12 entry return address stack. Internally, x86 instructions are translated into macro-ops, that are scheduled on one of the execution units. Results end up in the load-store units, either for the L1 D-cache (12 entries) for the L2 D-cache (32 entries).

On this platform we used 34 performance counter events to measure the execution events. These events can be roughly divided into six parts: (i) general (retired instructions etc.), (ii) processor frontend (I-cache and I-TLB events), (iii) branch prediction, (iv) processor core (stalls), (v) data cache, and (vi) memory requests seen on the bus. To do this, we used the perfctr kernel patch[4] for the 2.4.19 Linux kernel. The main advantage of using performance counters is that measuring is possible at native speed. The primary disadvantage is that the number of events that can be measured simultaneously is usually limited; in the case of the AMD Duron, only four events can be measured per run.

III. Statistical analysis

When measuring the execution of a benchmark application, with 34 performance counter events, the benchmark can be viewed as a single point in a 34-dimensional space, where each measured event represents a single dimension. It is very hard to understand the (dis)similarities between multiple points in this space because the different events are usually correlated to some degree. This means that e.g. the Euclidian distance isn't a reliable measure to determine (dis)similarity between two workloads.

[4]http://user.it.uu.se/ mikpe/linux/perfctr/

A. Principal Components Analysis

To improve the understanding on the relation between different workloads in the workload space, and to remove the correlation that exists between events, we use Principal Components Analysis (PCA) [JW02]. With PCA, new variables (Z_i) are computed from the original variables (X_j) (these are the events measured); the former are linear combinations of the latter. The transformation has the interesting properties that (i)

$$\mathrm{Cov}[Z_i, Z_j] = 0, i, j \in \{1, \ldots, n\}, i \neq j$$

and (ii) the variance exhibited by the $Z_i, i \in \{1, \ldots, n\}$ obeys the following rule:

$$\mathrm{Var}[Z_1] \geq \mathrm{Var}[Z_2] \geq \ldots \geq \mathrm{Var}[Z_n] \ .$$

This means that Z_1 accounts for the most variance, and thus for the most information, while Z_n accounts for the smallest amount of information. However, no information is lost, because

$$\sum_{i=1}^{n} \mathrm{Var}[X_i] = \sum_{i=1}^{n} \mathrm{Var}[Z_i] \ .$$

Usually the first few principal components account for 80% to 85% of the observed variance in the data. Because the variance exhibited by the last $n - p$ principal components is quite small, compared to the variance exhibited by the first p principal components, the number of components can be reduced by retaining only the first p components. This reduces the dimension of the space from n to p.

B. Cluster Analysis

Even for a small number of retained principal components, it can be quite hard to visualize the data points (just imagine plotting a 4D or a 6D space), and obtain a clear understanding of the (dis)similarities between these data points. To overcome this, we use Cluster Analysis (CA) [JW02]. The aim of CA is to cluster data points, where the most closely related points are clustered first. Each cluster can be represented by a new data point, that is then used to advance the clustering. We perform CA of the data points residing in the principal components space, which allows using the Euclidian distance to measure (dis)similarity between various data points.

We have used the linkage distance clustering algorithm which operates as follows. As a starting point for the algorithm, each Java workload is considered as a group. During each step of the analysis, two groups that lie most closely to each other, are bundled into a larger group. Thus groups are gradually clustered until we retain a single group. The decision on which groups to cluster is based solely on the (Euclidian) distance between two groups. We have used the pair-group average linkage distance, which effectively means that the distance between two groups is calculated as the average distance between all the members of the the groups considered.

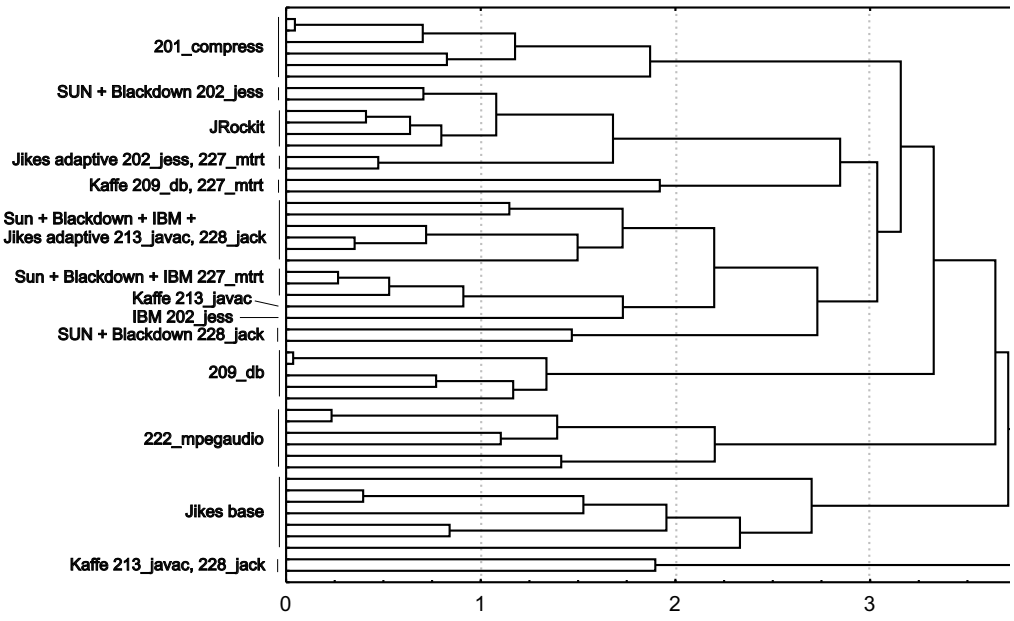

Fig. 1. Dendrogram for the SPECjvm98 s100 benchmark set, executed on seven virtual machine configurations

IV. Evaluation

In Figure 1 we show the dendrogram obtained after performing PCA and CA for the SPECjvm98 benchmark suite with the s100 input set. We observe two kinds of clusters, (i) benchmark clusters, where the main influence in directed by the application being run, and (ii) VM clusters, where the virtual machine has a large impact on the execution. For further results, we refer to the poster.

From our experiments, we conclude the following. In general, studies should use at least a few virtual machines in order to obtain reliable results, because the virtual machine itself

may have a significant influence, especially on shorter running applications. Deciding which benchmarks to use in any study is of course highly dependent on the topic that is studied, nevertheless our results give firm indications that it would be best to use a few benchmarks from multiple suites [EGB03].

REFERENCES

[AAA+00] B. Alpern, B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[Adv02] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, February 2002. `http://www.amd.com`.

[BEA02] BEA Systems, Inc. *BEA Weblogic JRockit—The Server JVM: Increasing Server-side Performance and Manageability*, August 2002. `http://www.bea.com/products/weblogic/jrockit`.

[EGB03] L. Eeckhout, A. Georges, and K. De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2003*, page *Accepted for publication*, October 2003.

[JW02] R. A. Johnson and D.W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time compiler. *IBM Systems Journal*, 39(1):175–193, February 2000.

[Sun02] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine, v1.4.1*, September 2002.