How Java Programs Interact with Virtual Machines at the Microarchitectural Level

Lieven Eeckhout Andy Georges Koen De Bosschere

Department of Electronics and Information Systems (ELIS), Ghent University
St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium

{leeckhou,ageorges,kdb}@elis.UGent.be

ABSTRACT

Java workloads are becoming increasingly prominent on various platforms ranging from embedded systems, over general-purpose computers to high-end servers. Understanding the implications of all the aspects involved when running Java workloads, is thus extremely important during the design of a system that will run such workloads. In other words, understanding the interaction between the Java application, its input and the virtual machine it runs on, is key to a succesful design. The goal of this paper is to study this complex interaction at the microarchitectural level, e.g., by analyzing the branch behavior, the cache behavior, etc. This is done by measuring a large number of performance characteristics using performance counters on an AMD K7 Duron microprocessor. These performance characteristics are measured for seven virtual machine configurations, and a collection of Java benchmarks with corresponding inputs coming from the SPECjvm98 benchmark suite, the SPECjbb2000 benchmark suite, the Java Grande Forum benchmark suite and an open-source raytracer, called Raja with 19 scene descriptions. This large amount of data is further analyzed using statistical data analysis techniques, namely principal components analysis and cluster analysis. These techniques provide useful insights in an understandable way.

From our experiments, we conclude that (i) the behavior observed at the microarchitectural level is primarily determined by the virtual machine for small input sets, e.g., the SPECjvm98 `s1` input set; (ii) the behavior can be quite different for various input sets, e.g., short-running versus long-running benchmarks; (iii) for long-running benchmarks with few hot spots, the behavior can be primarily determined by the Java program and not the virtual machine, i.e., all the virtual machines optimize the hot spots to similarly behaving native code; (iv) in general, the behavior of a Java application running on one virtual machine can be significantly different from running on another virtual machine. These conclusions warn researchers working on Java workloads to be careful when using a limited number of Java benchmarks or virtual machines since this might lead to biased conclusions.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: design studies, measurement techniques, performance attributes

General Terms

Measurement, Performance, Experimentation

Keywords

workload characterization, performance analysis, statistical data analysis, Java workloads, virtual machine technology

1. INTRODUCTION

In the last few years, the Java programming language is taking up a more prominent role in the software field. From high-end application servers, to webservers, to desktop applications and finally to small applications on portable or embedded devices, Java applications are used in virtually every area of the computing sector. Not only Java applications are abundant, the advent of the language also introduced various virtual machines capable of executing these applications, each with their own merits and drawbacks.

We can distinguish three important aspects that possibly have a large impact on the overall behavior of a Java workload: the virtual machine executing the Java bytecode, the Java application itself and the input to the Java application. For example, concerning the virtual machine, the choice of interpretation versus Just-in-Time (JIT) compilation is a very important one. Also, the mechanism implemented for supporting Java threads as well as for supporting garbage collection can have a large impact on the overall performance. Secondly, the nature of the Java application itself can have a large impact on the behavior observed by the microprocessor. For example, we can expect a database application to behave differently from a game application. Third, the input of the Java application can have a significant impact on the behavior of a Java workload. For example, a large input can cause a large number of objects being created during the execution of the Java application stressing the memory subsystem. Each of these three aspects can thus have a large impact on the behavior as observed at the microarchitectural level (in terms of branch behavior, cache behavior, instruction-level parallelism, etc.). This close in-

teraction between virtual machine, Java application and input is hard to understand due to the complex behavior of Java workloads. Therefore, we need techniques to get better insight in this interaction.

The main question we want to address in this paper is thus the following: how much of the behavior as observed at the microprocessor level is due to the virtual machine, the Java application, and the input to the application? For example, most virtual machines currently employ a JIT compilation/optimization strategy. But how big is the impact of the actual implementation of the JIT engine on the observed behavior? I.e., do virtual machines implementing more or less the same strategy behave similarly? Secondly, how large is the impact of the Java application? Is the behavior of a Java workload primarily determined by the Java application or by the virtual machine? And what is the impact of the input to the Java application?

In the last few years, valuable research has been done on characterizing Java workloads to get better insight in its behavior, see also the related work section at the end of the paper. Previous work typically considered only one or two virtual machines in their methodology as well as only one benchmark suite, mostly SPECjvm98. In addition, some studies use a small input set, e.g., `s1` for SPECjvm98, to limit the simulation time in their study. As such, we can raise the following questions in relation to previous work. Is such a methodology reliable for Java workloads? What happens if the behavior of a Java workload is highly dependent on the chosen virtual machine? Can we translate conclusions made for one virtual machine to another virtual machine? Also, is SPECjvm98 representative for other Java applications? I.e., are the conclusions taken based on SPECjvm98 valid for other Java programs? And is using a small input, e.g., SPECjvm98 `s1`, yielding a short-running Java workload representative for a large input, e.g., `s100`, yielding a long-running Java workload?

To answer these questions, we use the following methodology. First, we measure workload characteristics through performance counters while running the Java workloads on real hardware, in our case an AMD K7 Duron microprocessor. This is done for a large number of virtual machine configurations (7 in total) as well as for a large number of Java applications with corresponding inputs. The benchmarks and their inputs are taken from the SPECjvm98 suite, the SPECjbb2000 suite and the Java Grande Forum suite. In addition, we also include a raytracer with 19 scene descriptions. Second, a statistical analysis is done on these data using principal components analysis (PCA) [17]. PCA is a multivariate statistical data reduction technique capable of increasing the understandability of the large amounts of data. The basic idea of this statistical analysis is as follows. Java workloads could be displayed in a n-dimensional space, with n the number of performance characteristics measured in the previous step. However, the dimension of this space n is too large to be understandable, in our study $n = 34$. PCA reduces this high dimensional space to a lower dimensional and uncorrelated space, typically 4-D in our experiments without loosing important information. This increases the understandability for two reasons: (i) its lower dimension and (ii) there is no correlation between the axes in this space. In the third step of our methodology, we display the Java workloads in this lower dimensional space obtained after PCA. In addition, we further analyze this reduced Java workload space through cluster analysis (CA) [17]. This methodology will allow us to address the questions raised in this paper. Indeed, Java workloads that are far away from each other in this space show dissimilar behavior whereas Java workloads close to each other show similar behavior. As such, if Java workloads are clustered per virtual machine, i.e., all the Java applications running on one particular virtual machine are close to each other, we can conclude that the overall behavior is primarily determined by the virtual machine and not the Java application. Likewise, if Java workloads are clustered per Java application, we conclude that the Java application has the largest impact and not the virtual machine. Also, if a Java program running different inputs results in clustered data points, we can conclude that the input has a small impact on the overall behavior.

Answering the questions raised in this paper is of interest for various research domains. First, Java application developers can get insight in the behavior of the code they are developing and how their code interacts with the virtual machine and its input. For example, if the overall behavior is primarily influenced by the virtual machine and not the Java application, application developers will pay less attention to the performance of their code but will focus more on its reusability or reliability. Second, virtual machine developers will get better insight in what sense the behavior of a Java workload is influenced by the virtual machine implementation and more in particular, how Java programs interact with their virtual machine design. Using this information, virtual machine developers might design better VMs. Third, microprocessor designers can get insight in how Java workloads behave and how their microprocessors should be designed to address specific issues posed by Java workloads. Also, for microprocessor designers who heavily rely on time-consuming simulations, it is extremely useful to know whether small inputs result in similar behavior as large inputs and can thus be used to reduce the total simulation time without compromising the accuracy of their simulation runs [13].

This paper is organized as follows. In the next section, we present the experimental setup of this paper. We distinguish four components in our setup: (i) the Java workloads, consisting of the virtual machine, the Java benchmarks and if available, various inputs for each of these benchmarks; (ii) the hardware platform, namely the AMD K7 Duron microprocessor; (iii) the measurement technique, i.e., the use of on-chip performance counters; and (iv) the workload characteristics we use in our methodology. In section 3, we discuss the statistical data analysis techniques, namely principal components analysis (PCA) and cluster analysis (CA). In section 4 we present the results we obtain through our analysis and extensively discuss the conclusions that can be taken from these. Section 5 discusses related work on characterizing Java workloads. Finally, we conclude in section 6.

2. EXPERIMENTAL SETUP

2.1 Java workloads

This section discusses the virtual machines and the Java applications that are used in this study.

2.1.1 Virtual machines

In our study, we have used seven virtual machine configurations which are tabulated in Table 1: SUN JRE 1.4.1,

Blackdown JRE 1.4.1 Beta, IBM JRE 1.4.1, JikesRVM, JRockit and Kaffe.

Both the SUN JRE 1.4.1 and the Blackdown JRE 1.4.1 Beta virtual machines are based on the same SUN HotSpot virtual machine core [26]. HotSpot uses a mixed scheme of interpretation, Just-in-Time (JIT) compilation and optimization to execute Java applications. The degree of optimization can be specified by choosing either client mode or server mode. In client mode, the virtual machine performs fewer runtime optimizations resulting in a limited application startup time and a reduced memory footprint. In server mode, the virtual machine performs classic code optimizations as well as optimizations that are more specific to Java, such as null-check and range-check elimination. It is also interesting to note that HotSpot maps Java threads to native OS threads. The garbage collector uses a fully accurate, generational copying scheme. New objects are allocated in the 'nursery' and moved to the 'old object' space when the 'nursery' is collected. Objects in the 'old object' space are reclaimed by a mark and sweep compacting strategy.

BEA Weblogic's JRockit [6] is a virtual machine that is targeted at server-side Java. JRockit compiles methods upon their first invocation. At runtime, statistics are gathered and hot methods are scheduled for optimization. The optimized code replaces the old code while the virtual machine keeps running. This way, an adaptive optimization scheme is realized. JRockit uses a mixed threading scheme, called ThinThread, in which n Java threads are multiplexed on m native threads. The virtual machine comes with four possible garbage collection strategies. We have used the generational copying version in our experiments, which is the default for heap sizes less than 128MiB.

Jikes [2, 3] is a Research Virtual Machine (RVM)—previously known as Jalapeño—that is targeted at server-side Java applications. Jikes is written entirely in Java and uses compilation throughout the entire execution (no interpretation). It is possible to configure the JikesRVM in different compiling modes: baseline compiler, optimizing compiler and adaptive compiler. We have used the baseline and adaptive modes in our experiments. The threading system multiplexes n Java threads to m native threads. There is a range of garbage collection strategies available for this virtual machine. Among them are copying, mark-and-sweep and generational collectors as well as combinations of these strategies. We have used the non-generational copying scheme (SemiSpace).

Kaffe[1] is an open source virtual machine. We have used version 1.0.7 in our experiments. Kaffe uses interpretation as well as JIT compilation. In addition, native threads can be used.

The IBM JRE 1.4.0[2] [25] also uses a mixed strategy by employing IBM's JIT compiler as well as IBM's Mixed Mode Interpreter (MMI).

Note that the choice of the garbage collector is not consistent over the virtual machine configurations. We have chosen the default garbage collector for each virtual machine. This leads to different garbage collector mechanisms for different virtual machines as can be seen from Table 1. In section 4.4, we will evaluate the impact of the garbage collector on overall workload behavior. This evaluation will show that the choice of the garbage collector has a minor impact on the results of this paper and does not change the overall conclusions.

[1] http://www.kaffe.org
[2] http://www.ibm.com

2.1.2 Java applications and their inputs

There are numerous Java applications available both in the public and the commercial domain. However, most of these are (highly) interactive. Using such applications for our purposes is unsuitable since the measurements would not be reproducable. As such, we used non-interactive Java programs with command line inputs. The applications we have used are taken from several sources, see also Table 2: SPECjvm98, SPECjbb2000, the Java Grande Forum suite, and Raja.

SPECjvm98[3] is a client-side Java benchmark suite consisting of seven benchmarks. For each of these, SPECjvm98 provides three inputs: `s1`, `s10` and `s100`. Contradictory to what the input set names suggest, the size of the input set does not increase linearly. For some benchmarks, a larger input indeed increases the problem size. For other benchmarks, a larger input executes a smaller input multiple times. In the evaluation section, we will discuss the impact of the various input sets on the behavior of the Java programs and their virtual machines. SPECjvm98 was designed to evaluate combined hardware (CPU, caches, memory, etc.) and software aspects (virtual machine, kernel activity, etc.) of a Java environment. However, they do not include graphics, networking or AWT (window management).

SPECjbb2000 (Java Business Benchmark)[4] is a server-side benchmark suite focussing on the middle-tier, the business logic, of a three-tier system. We have run the SPECjbb2000 benchmark with different numbers of warehouses: 2, 4 and 8 warehouses.

The Java Grande Forum (JGF) benchmark suite[5] [9] is intended to study the performance of Java in the context of so-called *Grande* applications, i.e., applications requiring large amounts of memory, bandwidth and/or processing power. Examples include computational science and engineering codes, large scale database applications as well as business and financial models. For this paper, we have chosen four large scale applications from the sequential suite which are suitable for uniprocessor performance evaluation. For each of these benchmarks, we have used the two available problem sizes, small and large.

Raja[6] is a raytracer in Java. We included this raytracer in our analysis since its distribution comes with 19 scene descriptions. As such we will be able to quantify the impact of the input on the behavior of the raytracer. Unfortunately, we were unable to execute this benchmark on the Jikes and Kaffe virtual machines.

We ran all the benchmarks with a standard 64MiB virtual machine heap size. For SPECjbb2000, we used a heap size of 256 MiB.

2.2 Hardware used

We have done all our experiments on a x86-compatible platform, namely a 1GHz AMD Duron (model 7). The microarchitecture of the AMD Duron is identical to the AMD Athlon's microarchitecture except for the reduced size of the

[3] http://www.spec.org/jvm98
[4] http://www.spec.org/jbb2000
[5] http://www.javagrande.org
[6] http://raja.sourceforge.net

Virtual machine	Configuration used
SUN JRE 1.4.1	HotSpot client, generational non-incremental garbage collection
Blackdown JRE 1.4.1	HotSpot client, generational non-incremental garbage collection
JikesRVM base	baseline compiler with copying garbage collection
JikesRVM adpt	adaptive compiler with copying garbage collection
JRockit	adaptive optimizing compiler, generational copying collector
Kaffe	interpretation and JIT compilation, non-generational garbage collection
IBM JRE 1.4.1	interpretation and JIT compilation

Table 1: The virtual machine configurations we have used to perform our measurements.

SPECjvm98	
201_compress	A compression program, using a LZW method ported from 129.compress in the SPECCPU95 suite. Unlike 129.compress, it processes real data from several files. The various inputs are obtained by performing a different number of iterations through various input files. It requires a heap size of 20MiB and allocates 334MiB of objects.
202_jess	An expert shell system, adapted from the CLIPS system. The various inputs consist of a set of puzzles to be solved, with varying degrees of difficulty. The benchmark requires a heap size of 2MiB while allocating 748MiB of objects.
209_db	The benchmark performs a set of database requests on a memory resident database of 1MiB. The various inputs are obtained by varying the number of requests to the database. It requires a heap size of 16MiB and allocates 224MiB of objects.
213_javac	This is the JDK 1.0.2 source code compiler. The various inputs are obtained by making multiple copies of the same input files. It requires a heap size of 12MiB, and allocates 518MiB of objects.
222_mpegaudio	A commercial application decompressing MPEG Layer-3 audio files. The input consists of about 4MiB of audio data. The number of objects that are allocated is negligible.
227_mtrt	A raytracer using two threads to render a scene. The various inputs are determined by the problem size. The benchmark requires a heap size of 16MiB and allocates 355MiB of objects.
228_jack	An early version of JavaCC which is a Java parser generator. The various inputs make several passes through the same data. Execution requires a heap size of 2MiB while 481MiB of objects are allocated.
SPECjbb2000	A three-tier transaction system, where the user interaction is simulated by random input selection and the third tier, the database, is represented by a set of binary trees. The benchmark focuses on the business logic found in the middle tier. It is loosely based on the IBM pBOB benchmark [5]. About 256MiB of heap space is required to run the benchmark.
Java Grande Forum	
search	A program solving a connect-4 game, using an alpha-beta pruning technique. The problem size is determined by the starting position from which the game is solved. The heap size should be at least 6MiB for both inputs.
euler	Solution for a set of time-dependent Euler equations modeling a channel with a bumped wall, using a fourth order Runge-Kutta scheme. The model is evaluated for 200 timesteps. The problem size is determined by the size of the mesh on which the solution is computed. The heap size that is required is 8MiB for the small input and 15MiB for the large input.
moldyn	Evaluation of an N-body model for particles interacting under a Lennard-Jones potential in a cubic space. The problem size is determined by the number of particles. Both inputs need a heap size of 1 MiB.
raytracer	A raytracer rendering a scene containing 64 spheres. The problem size is determined by the resolution of the rendered image. Both inputs require a heap size of 1 MiB.
Raja	A Raytracer. We used the latest 0.4.0-pre4 version. Input variation is obtained by using a set of 19 scene descriptions.

Table 2: The benchmarks we used in our measurements.

component	subcomponent	description
memory hierarchy	L1 I-cache	64KB two-way set-associative, 64-byte lines, LRU replacement with next line prefetching
	L1 D-cache	64KB two-way set-associative, 8 banks with 8-byte lines, LRU write-allocate, write-back, two access ports 64 bits each
	L2 cache	64KB two-way set-associative, unified, on-chip, exclusive
	L1 I-TLB	24 entries, fully associative
	L2 I-TLB	256 entries, four-way set-associative
	L1 D-TLB	32 entries, fully associative
	L2 D-TLB	256 entries, four-way set-associative
branch prediction	BTB	branch target buffer, two-way set-associative, 2048 entries
	RAS	return address stack, 12 entries
	taken/not-taken	gshare 2048-entry branch predictor with 2-bit counters
system design	bus	200MHz, 1.6GiB per second
pipeline stages	integer	10 cycles
	floating-point	15 cycles
integer pipeline	pipeline 1	integer execution unit and address generation unit also allows integer multiply
	pipeline 2	integer execution unit and address generation unit
	pipeline 3	idem
floating-point pipeline	pipeline 1	3DNow! add, MMX ALU/shifter and floating-point add
	pipleine 2	3DNow!/MMX multiply/reciproce, MMX ALU and floating-point multiply/divide/square root
	pipeline 3	floating-point constant loads and stores

Table 3: The AMD K7 Duron microprocessor summary.

L2 cache (64KB instead of 256KB). As such, the Duron as well as the Athlon belong to the same AMD K7 processor family [1, 12]. For more details on the AMD Duron that is used in this study we refer to Table 3. The AMD K7 is a superscalar microprocessor implementing the IA-32 instruction set architecture (ISA). It has a pipelined microarchitecture in which up to three x86 instructions can be fetched. These instructions are fetched from a large predecoded 64KB L1 instruction cache (I-cache). For dealing with the branches in the instruction stream, branch prediction is done using a global history (gshare) based taken/not-taken branch predictor, a branch target buffer (BTB) and a return address stack (RAS). Once fetched, each (variable-length) x86 instruction is decoded into a number of simpler (and fixed-length) macro-ops. Up to three x86 instructions can be translated per cycle.

These macro-ops are then passed to the next stage in the pipeline, the instruction control unit (ICU) which basically consists of a 72-entry reorder buffer. From this reorder buffer, macro-ops are scheduled into an 18-entry integer scheduler and a 36-entry floating-point scheduler for integer and floating-point operations, respectively. The 18-entry integer scheduler is organized as a collection of three 6-entry deep reservation stations, each reservation station serving an integer execution unit and an address generation unit. The 36-entry floating-point scheduler (FPU: floating-point unit) serves three floating-point pipelines executing x87, MMX and 3DNow! operations. In the schedulers, the macro-ops are broken down to ops which can execute out-of-order. Next to these schedulers, the AMD K7 microarchitecture also has a 44-entry load-store unit. The load-store unit consists of two queues, a 12-entry queue for L1 D-cache load and store accesses and a 32-entry queue for L2 cache and memory load and store accesses—requests that missed in the L1 D-cache. The L1 D-cache is organized as an eight-bank cache having two 64-bit access ports.

Another interesting aspect of the AMD K7 microarchitecture is the fact that the L2 unified cache is an exclusive cache. This means that cache blocks that were previously held by the L1 caches but had to be evicted from L1, are held in L2. If the newer cache block that is to be stored in L1 previously resided in L2, that cache block will be evicted from L2 to make room for the L1 block, i.e., a swap operation is done between L1 and L2. If the newer cache block that is to be stored in L1 did not previously reside in L2, a cache block will need to be evicted from L2 to memory.

2.3 Performance counters

The AMD K7 Duron has a set of microprocessor-specific registers. These registers can be used to obtain information about the processor's usage during the execution of a computer program. This kind of information is held in so called *performance counter* registers. We have used the performance counter registers available in the AMD Duron to measure several characteristics of benchmark executions. Performance counters have several important benefits over alternative characterization methods. First, characteristics are obtained very fast since we run computer programs on native hardware. Alternative options are significantly less efficient. For example, measuring characteristics using instrumented binaries inevitably results in a serious slowdown. Measuring characteristics through simulation is even worse since detailed simulation is approximately a factor 100,000 slower than native execution. The second advantage of using performance counters is that setting up the infrastructure for doing these experiments is extremely simple: no simulators, nor instrumentation routines have to be written. Third, measuring kernel activity using performance

counters comes for free. Instrumentation or simulation on the other hand, require either instrumenting kernel code or employing a full system simulator. The fourth advantage of performance counters is that characteristics are measured on real hardware instead of a software model. The latter can lead to inaccuracies due to its higher abstraction level [11].

Unfortunately, performance counters also come with their disadvantages. First, measuring an event of two executions of the same computer program can lead to slightly different results. One reason for this is cache contention due to multitasking, interrupts, etc. To cope with this problem, each event can be measured multiple times and an average number of these measurements can be used throughout the analysis. For this study, we have measured each event four times and the arithmetic average is used in the analysis. A second problem with performance counters is that only a limited number of events can be measured per program execution, e.g., four events for the AMD K7. As such, to measure the 34 events as listed in Table 4 we had to run each program nine times. Note that these two slowdown factors result in the fact that each program needs to be run 36 times, i.e., 4 times for making the average over four program runs multiplied by 9 times for measuring all events (4 events per program run). As such, using the approach of performance counters, although running on native hardware, yields a slowdown of a factor 36 over one single native program execution. Note that this is still much faster than through instrumentation (slowdown factor heavily depending on the instrumentation routines, typically more than 1,000) or simulation (slowdown factor of 50,000 up to 300,000 [4, 7]). A third disadvantage of performance counters is that the sensitivity to performance of a microarchitectural parameter cannot be measured since the microarchitecture is fixed. This disadvantage could be remedied by measuring characteristics on multiple platforms having different microprocessors.

In our environment, reading the contents of the performance counter registers is done using the perfctr version 2.4.0 package[7] which provides a patch to the most common Linux/x86 kernels. Our Linux/x86 evironment is RedHat 7.3 with kernel 2.4.19-11. The perfctr package keeps track of the contents of the performance counter registers on a per-process basis. This means that the contents of the performance counters are saved on a context switch and restored after the context switch. This allows precise per-process measurements on a multi-tasking operating system such as Linux. In order to use this package for our purpose we had to extend the perfctr package to deal with multi-threaded Java. The original perfctr package v2.4.0 is only capable of measuring the performance counter values for a single-threaded process. However, in most modern virtual machines running Java applications, all the Java threads are actually run as native threads or (under Linux) separate processes. Other VMs multiplex their n Java threads on a set of m native threads, for example JRockit [6] and Jikes [2, 3]. Yet other VMs map all Java threads to a single native thread. In this case, the Java threads are often called *green* threads. To be able to measure the characteristics for all the threads running in a virtual machine that uses multiple native threads, we extended the perfctr package. This way, all the Java threads that are created during the execution of a Java application are profiled.

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

2.4 Workload characteristics

The processor events that were measured for this study on the AMD Duron are tabulated in Table 4. These 34 workload characteristics can be roughly divided in six groups:

- **General characteristics**. This group of events contains the number of clock cycles needed to execute the application; the number of retired x86 instructions; the number of retired operations—recall that x86 instructions are broken down to fixed-length and much simpler operations; the number of retired branches, etc.

- **Processor frontend**. Here we have grouped characteristics that are related to the processor frontend, i.e., the I-cache and the fetch unit: the number of fetches from the L1 I-cache, the number of L1 I-cache misses, the number of instruction fetches from the L2 instruction cache and the number of instruction fetches from main memory. Next to these characteristics, we also measure the L1 I-TLB misses that hit the L2 TLB, as well as the L1 I-TLB misses that also miss the L2 I-TLB. In addition, we also measure the number of fetch unit stall cycles.

- **Branch prediction**. This group measures the performance of the branch prediction hardware: the number of branch taken/not-taken mispredictions, the number of branch target mispredictions, the performance of the return address stack (RAS), etc.

- **Processor core**. The performance counters that deal with the processor core basically measure stall cycles, i.e., cycles in which no new instructions can be further pushed down the pipeline due to data, control or structural hazards, for example, due to a read-after-write dependency, an unavailable functional unit, an unresolved D-cache miss, a branch misprediction, etc. In this group we make a distinction between the following events: an integer control unit (ICU) full stall, a reservation station full stall, a floating-point unit (FPU) full stall, load-store unit queue full stalls, and a dispatch stall which can be the result of a number of combined stall events.

- **Data cache**. We distinguish the following characteristics related to the data cache: the number of L1 D-cache accesses, the number of L1 D-cache misses, the number of refills from L2, the number of refills from main memory and the number of writebacks. We also measure the L1 D-TLB misses that hit the L2 D-TLB and the L1 D-TLB misses that also miss the L2 D-TLB.

- **Bus unit**. We monitor the number of requests to the main memory, as seen on the bus.

The performance characteristics that are actually used in the statistical analysis, are all divided by the number of clock cycles. By doing so, the events are actually measured per unit of time. For example, one particular performance characteristic will be the number of L1 D-cache misses per unit of time, in casu, per clock cycle. Note that this performance measure is more appropriate than the L1 D-cache miss rate, often used in other studies, since it is more directly related

component	abbrev.	decription
general	cycles	number of clock cycles
	instr	number of retired x86 instructions
	ops	number of retired operations
	br	number of retired branches
	br_taken	number of retired taken branches
	far_ctrl	number of retired far control instructions
	ret	number of retired near return instructions
processor frontend	ic_fetch	number of L1 I-cache fetches
	ic_miss	number of L1 I-cache misses
	ic_L2_fetch	number of L2 instruction fetches
	ic_mem	number of instruction fetches from memory
	itlb_L1_miss	number of L1 I-TLB misses, but L2 I-TLB hits
	itlb_L2_miss	number of L1 and L2 I-TLB misses
	fetch_stall	number of fetch unit stall cycles
branch prediction	br_mpred	number of retired mispredicted branches
	br_taken_mpred	number of retired mispredicted taken branches
	ret_mpred	number of retired mispredicted near return instructions
	target_mpred	number of mispredicted branches due to address miscompare
	ras_hits	number of return address stack hits
	ras_oflow	number of return address stack overflows
processor core	dispatch_stall	number of dispatch stall cycles (combined stall events)
	icu_full	number of integer control unit (ICU) full stall cycles
	res_stat_full	number of reservation station full stall cycles
	fpu_full	number of floating-point unit (FPU) full stall cycles
	lsu_full	number of load-store unit (LSU) full stall cycles concerning the L1 D-cache access queue
	lsu_L2_full	number of load-store unit (LSU) full stall cycles concerning the L2 and memory access queue
data cache	dc_access	number of L1 data cache accesses equals number of load-store operations
	dc_miss	number of L1 data cache misses
	dc_L2	number of refills from the L2 cache
	dc_mem	number of refills from main memory
	dc_wb	number of writebacks
	dtlb_L1_miss	number of L1 D-TLB misses, but L2 D-TLB hits
	dtlb_L2_miss	number of L1 and L2 D-TLB misses
system bus	mem_requests	number of memory requests as seen on the bus

Table 4: The 34 workload characteristics obtained from the performance counters on the AMD Duron.

to actual performance. Indeed, a high D-cache miss rate can still result in a low number of D-cache misses per unit of time if the number of D-cache accesses is low.

As stated in the previous section, performance counters can be measured for both kernel and user activity. Since it is well known from previous work [19] that Java programs spend a significant amount of time in kernel activity, we have measured both.

3. STATISTICAL ANALYSIS

From the previous sections it becomes clear that the amount of data that is obtained from our measurements is huge. Indeed, each performance counter event is measured for each benchmark, for each virtual machine and for each input. As such, the total amount of data is too large to be analyzed understandably. In addition, there exists correlation between the various events which makes the interpretation of the data even more difficult for the purpose of this paper. Therefore, we use a methodology [13, 14] that is based on statistical data analysis, namely principal components analysis (PCA) and cluster analysis (CA) [17], to present a different view on the measured data. Applying these statistical analysis techniques was done using the commercial software package STATISTICA [24]. We will discuss PCA and CA in the following two subsections.

3.1 Principal components analysis

The basic idea of our approach is that a Java workload—a Java workload is determined by the Java application, its input and the virtual machine—could be viewed as a point in the multidimensional space built up by the performance counter events. Before applying any statistical analysis technique, we first normalize the data, i.e., mean and variance of each event is zero and one, respectively. Subsequently, we apply principal components analysis (PCA) which transforms the data into uncorrelated data. This is beneficial for our purpose of measuring (dis)similarity between Java workloads. Measuring (dis)similarity between two Java workloads based on the original non-normalized and correlated events on the other hand, would give a distorted view. In-

deed, the Euclidean distance between two Java workloads in the original space is not a reliable measure for two reasons. First, non-normalized data gives a higher weight to events with a higher variance. Through normalization, all events get equal weights. Second, the Euclidean distance in a correlated space gives a higher weight to correlated variables. Since correlated variables in essence measure the same underlying program characteristic, we propose to remove that correlation through PCA.

PCA computes new variables, called *principal components*, which are *linear combinations* of the original variables, such that all principal components are uncorrelated. PCA tranforms the p variables $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$ which means that Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ which means that there is no information overlap between the principal components. Note that the total variance in the data remains the same before and after the transformation, namely $\sum_{i=1}^{p} Var[X_i] = \sum_{i=1}^{p} Var[Z_i]$.

As stated in the first property in the previous paragraph, some of the principal components will have a high variance while others will have a small variance. By removing the components with the lowest variance from the analysis, we can reduce the number of program characteristics while controlling the amount of information that is thrown away. We retain q principal components which is a significant information reduction since $q \ll p$ in most cases, for example $q = 4$. To measure the fraction of information retained in this q-dimensional space, we use the amount of variance $(\sum_{i=1}^{q} Var[Z_i])/(\sum_{i=1}^{p} Var[X_i])$ accounted for by these q principal components. Typically 85% to 90% of the total variance should be explained by the retained principal components.

In this study the p original variables are the events measured through the performance counters, see section 2.4. By examining the most important q principal components, which are linear combinations of the original performance events ($Z_i = \sum_{j=1}^{p} a_{ij} X_j, i = 1, \ldots, q$), meaningful interpretations can be given to these principal components in terms of the original program characteristics. A coefficient a_{ij} that is close to +1 or -1 implies a strong impact of the original characteristic X_j on the principal component Z_i. A coefficient a_{ij} that is close to 0 on the other hand, implies no impact.

The next step in the analysis is to display the various Java workloads as points in the q-dimensional space built up by the q principal components. As such, a view can be given on the Java workload space. Note again that the projection on the q-dimensional space will be much easier to understand than a view on the original p-dimensional space for two reasons: (i) q is much smaller than p: $q \ll p$, and (ii) the q-dimensional space is uncorrelated.

3.2 Cluster analysis

Cluster analysis (CA) [17] is another data analysis technique that is aimed at clustering the Java workloads into groups that exhibit similar behavior. This is done based on a number of variables, in our case the principal components obtained from PCA. A commonly used algorithm for doing cluster analysis is *linkage clustering* which starts with a matrix of distances between the Java workloads. As a starting point for the algorithm, each Java workload is considered as a group. In each iteration of the algorithm, the two groups that are most close to each other (with the smallest distance, also called the *linkage distance*) will be combined to form a new group. As such, close groups are gradually merged until finally all cases will be in a single group. This can be represented in a so called *dendrogram*, which graphically represents the linkage distance for each group merge in each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to consider. This decision can be made based on the linkage distance. Indeed, small linkage distances imply strong clustering while large linkage distances imply weak clustering. Their exist several methods for calculating the distance between two groups. In this paper, we have used the *pair-group average* strategy. This means that the distance between two groups is defined as the average distance between all the members of each group.

The reason why we chose to first perform PCA and subsequently cluster analysis instead of applying cluster analysis on the initial data is as follows. The original variables are highly correlated which implies that an Euclidean distance in this space is unreliable due to this correlation as explained previously. First performing PCA alleviates this problem. In addition, PCA gives us the opportunity to visualize and understand why two Java workloads are different from each other.

4. EVALUATION RESULTS

In this evaluation section, we present and extensively discuss the results that were obtained from our analysis. First, we present the results for the `s1` and `s100` input sets of the SPECjvm98 benchmark suite. Second, we analyze the behavior of the Java Grande Forum workloads. And finally, we present the complete picture with all the Java workloads considered in this study. We present the results for the SPECjvm98 benchmark and the Java Grande Forum before presenting the complete picture for several reasons. First, it makes the results obtained in this paper more comparable to previous work mostly done on SPECjvm98. Second, it makes the understanding easier by building up the complexity of the data. Third, it allows us to demonstrate the relativity of this methodology. In other words, the results obtained from PCA or CA quantify the (dis)similarity between the Java workloads included in the analysis, but say nothing about the behavior of these workloads in comparison to other Java workloads not included in the analysis.

4.1 SPECjvm98

The SPECjvm98 benchmark suite offers three input sets, commonly referred to as the `s1`, `s10` and `s100` input set. All these benchmarks are executed using the virtual machines summarized in Table 1, with a maximal heap size of 64MiB. We first discuss the results of the `s1` input set after which we discuss the results for `s100`.

4.1.1 Analysis of the s1 input set

For the data with the `s1` input set, we retain four principal components that account for 86.5% of the observed variance in the measurements of the 49 Java workloads (7 SPECjvm98 benchmarks times 7 VM configurations). The factor loadings obtained for the principal components are given in Figure 1. These factor loadings account for 46.1%,

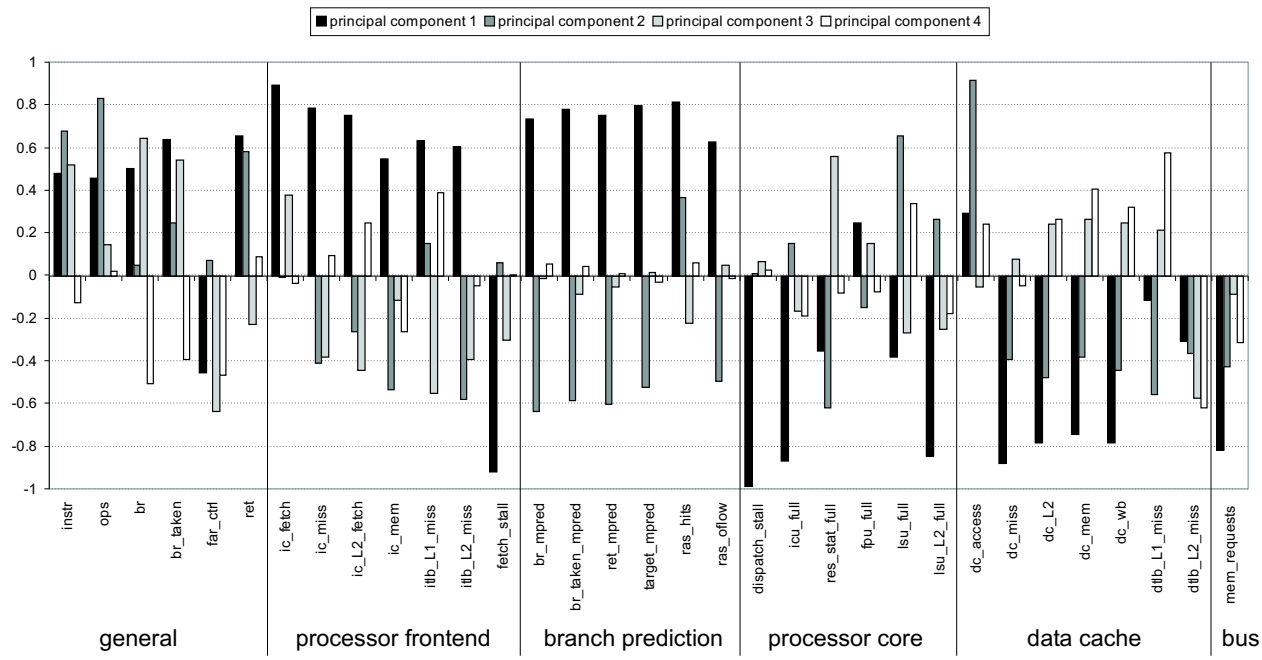

Figure 1: Factor loading for SPECjvm98 with the s1 input set.

22.2%, 11.1% and 7.2% of the total variance, respectively. When we take a closer look to the factor loadings a_{ij} of the retained principal components, it is obvious that the first component is by far the most important one. The contributions of the measured characteristics to the second, the third and fourth component are relatively smaller. In the following enumeration, we discuss the contributions made by each of the performance characteristics to each principal component:

- The main positive influence on the first principal component (PC_1) is caused by the branch prediction characteristics and processor frontend characteristics, with except for the amount of fetch stalls, see Table 4. The first principal component is negatively influenced by several stall events, i.e., the amount of fetch stalls, dispatch stalls, ICU full stalls and L2/memory LSU full stalls. In addition, PC_1 is also negatively affected by the number of data cache misses, data cache write-backs and data cache refills from L2 and from memory. Finally, PC_1 is also negatively influenced by the amount of memory requests seen on the bus.

- The second principal component (PC_2) is positively influenced by the number of x86 instructions retired per clock cycle and the number of retired operations per cycle, the amount of retired near return instructions, the number of stalls caused by a full L1 LSU unit, and the amount of data cache accesses. This component is negatively influenced by the number of instruction fetches from memory, by the number of L2 I-TLB misses, by the branch prediction accuracy and by the number of stalls caused by full reservation stations. It is also negatively influenced by the number of L1 D-TLB misses.

- For the third principal component (PC_3), we see that the amount of (taken) branches as well as the number of stalls caused by full reservation stations deliver the major positive contributions. PC_3 is negatively influenced by the amount of retired far control instructions, the amount of L1 I-TLB misses that hit the L2 I-TLB, and the amount of L2 D-TLB misses.

- The fourth principal component (PC_4) is the positively dominated by the amount of L1 D-TLB misses that hit in the L2 D-TLB, and negatively dominated by the amount of branches and the amount of L2 D-TLB misses.

The factor loadings also give an indication of the correlated characteristics for this set of Java workloads. For example, from these results we can conclude that (along the first principal component) the branch characteristics correlate well with the frontend characteristics. Moreover, this correlation is a positive correlation since both characteristics have a positive contribution to the first principal component. Also, the frontend characteristics correlate negatively with the amount of fetch stalls. In other words, this implies for example that a high number of I-cache fetches per unit of time correlates well with a low number of fetch stalls per unit of time which can be understood intuitively.

We can now display these Java workloads in the 4-dimensional space built up by the four principal components. This is shown in Figures 2 and 3 for the first versus the second principal component and the third versus the fourth principal component, respectively. Since we are dealing with a four-dimensional space, it is important to consider these two plots simultaneously to get a clear picture of the four dimensions. Note that in Figures 2 and 3, different SPECjvm98 benchmarks running on the same virtual machine are all represented by the same symbol. These graphs should be

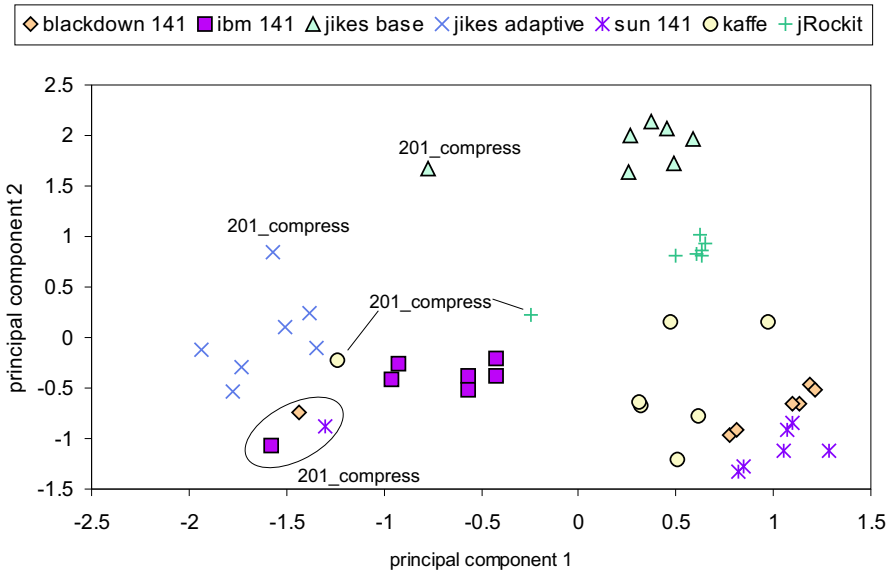

Figure 2: Scatterplot for the SPECjvm98 s1 workload set, as a function of the first and the second principal component. Different SPECjvm98 benchmarks running on the same virtual machine are represented by the same symbol.

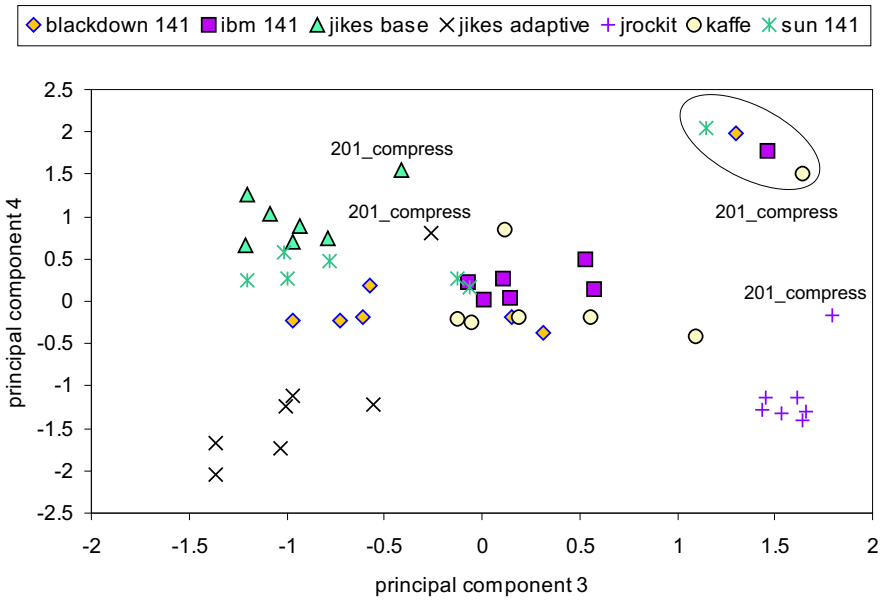

Figure 3: Scatterplot for the SPECjvm98 s1 workload set, as a function of the third and the fourth principal component. Different SPECjvm98 benchmarks running on the same virtual machine are represented by the same symbol.

interpreted as follows. A Java workload having a high coefficient along the first principal component shows a behavior that can be characterized by, see also Figure 1, high numbers for the branch characteristics and the frontend characteristics. In addition, low numbers will be observed for several stall characteristics (fetch, dispatch, ICU and L2/memory LSU), the number of data cache misses, the number of data refills from L2 and memory, the number of data writebacks, and the number of memory requests from the bus.

The graphs in Figures 2 and 3 clearly show that the data points are more or less clustered per virtual machine. Indeed, we observe tight clusters for JRockit, the baseline version of Jikes, the adaptive version of Jikes and the IBM 1.4.1 VM. The clusters corresponding to the SUN 1.4.1 VM and the Blackdown 1.4.1 VM, are clustered less tightly. Notice also that these two clusters are quite close to each other. This is obviously due to the fact that both virtual machines are built around the same HotSpot virtual machine core. This graph also reveals that Kaffe exhibits the least tightly clustered behavior. *From these results we can conclude that for the* s1 *input set, the virtual machine has a larger impact on the overall behavior than the Java application.* In other words, a virtual machine running a Java application with a small input will exhibit similar behavior irrespective of the Java application it is running. This can be understood intuitively since the s1 input set results in very short running benchmarks (in the order of seconds) for which the startup time of the virtual machine (initializing and loading significant parts of the JDK library) is the highest factor contributing to the overall behavior. *From these data we can also conclude that using the* s1 *input set of SPECjvm98 in a performance analysis might not be a good method unless one is primarily interested in measuring startup times, not just long-running performance.*

It is also interesting to note that the data points corresponding to the 201_compress benchmark are not part of the clusters discussed in the previous paragraph. In other words, for this Java benchmark, the interaction between the application and the virtual machine has a large impact on its overall behavior at the microarchitectural level since the various virtual machines for 201_compress are spread over the Java workload space. A close inspection of 201_compress reveals that it has a small code size, while processing a fairly large amount of data, even in case of the s1 input set. Profiling shows that for this benchmark, the top 10 methods that are called, account for 98% of all method calls. Clearly, 201_compress has a small number of hot methods, much smaller than the other SPECjvm98 benchmarks. This leads to a small working set and allows fairly aggressive optimizations by the virtual machine's native code generator. Since each virtual machine implements its run-time optimizer in a different way, this can result in a behavior that is quite different for each virtual machine. Note however that the SUN 1.4.1 VM, the Blackdown 1.4.1 VM and the IBM 1.4.1 VM yield quite similar behavior for 201_compress.

Another way of visualizing the (dis)similarity in this transformed space after PCA can be obtained through cluster analysis (CA). A dendrogram can be displayed which graphically represents the linkage distance during CA. This dendrogram is shown in Figure 4. In a dendrogram, data points connected through small linkage distances are clustered in early iterations of the algorithm and thus exhibit similar behavior. In our case, Java workloads exhibiting similar behavior will thus be connected through small linkage distances. Based on Figure 4, we can make the same conclusions as we made based on the visualization of the reduced space obtained after PCA, see Figures 2 and 3. For example, we clearly observe the four tight clusters per virtual machine: (i) the baseline Jikes virtual machine, (ii) the adaptive Jikes virtual machine, (iii) the JRockit virtual machine, and (iv) the IBM 1.4.1 virtual machine. Also, we clearly observe that the SUN 1.4.1 and the Blackdown 1.4.1 VMs are loosely clustered. In addition, the Kaffe virtual machine results in the least tight cluster. Finally, concerning 201_compress, we observe that the Java workloads are linked through large linkage distances, and that a tight cluster is observed for the SUN 1.4.1 VM, the IBM 1.4.1 VM and the Blackdown 1.4.1 VM running 201_compress.

4.1.2 Analysis of the s100 input set

For the s100 input set, we retain six principal components after PCA that account for 87.3% of the observed variance in the measurements. These six principal components account for 48.4%, 16.3%, 8.1%, 6.5%, 4.3% and 3.7% of the total variance, respectively. Note that the first four components account for 79.2% of the variance which is less than the variance explained by the four principal components for s1. This indicates that the data for s100 are not as much correlated as for s1. The factor loadings have the following contibutions from the various characteristics.

- For the first principal component (PC_1), there are positive contributions, mainly from the number of retired x86 instructions per cycle, the number of L1 and L2 I-cache fetches, the branch prediction accuracy, and the number of D-cache accesses. Negative contributions come from the number of fetch stalls and dispatch stalls, the number of D-cache misses, the number of D-cache writebacks and the number of requests made to memory as seen on the bus.

- For the second principal component (PC_2), positive contributions are made by the number of FPU full stalls, the amount of D-cache accesses, and the number of x86 retired instructions per cycle; while negative contributions are made by the branch prediction accuracy and the number of L1 D-cache misses.

- For the third principal component (PC_3), there is a single important positive contribution made by the number of branches. A negative contribution is made by the number of return address stack (RAS) overflows and the number of L1 LSU full stalls.

- The fourth component is positively influenced by the number of L1 D-TLB misses and the number of retired far control transfers. It is negatively influenced by the number of mispredicted indirect branches, the number of mispredicted near returns and the number of RAS overflows.

- The fifth component is positively dominated by the number of instruction fetches from memory and negatively dominated by the number of ICU full stalls.

- The sixth and last retained principal component is positively influenced by the number of I-fetches from the L2 cache and the number of L1 I-cache misses.

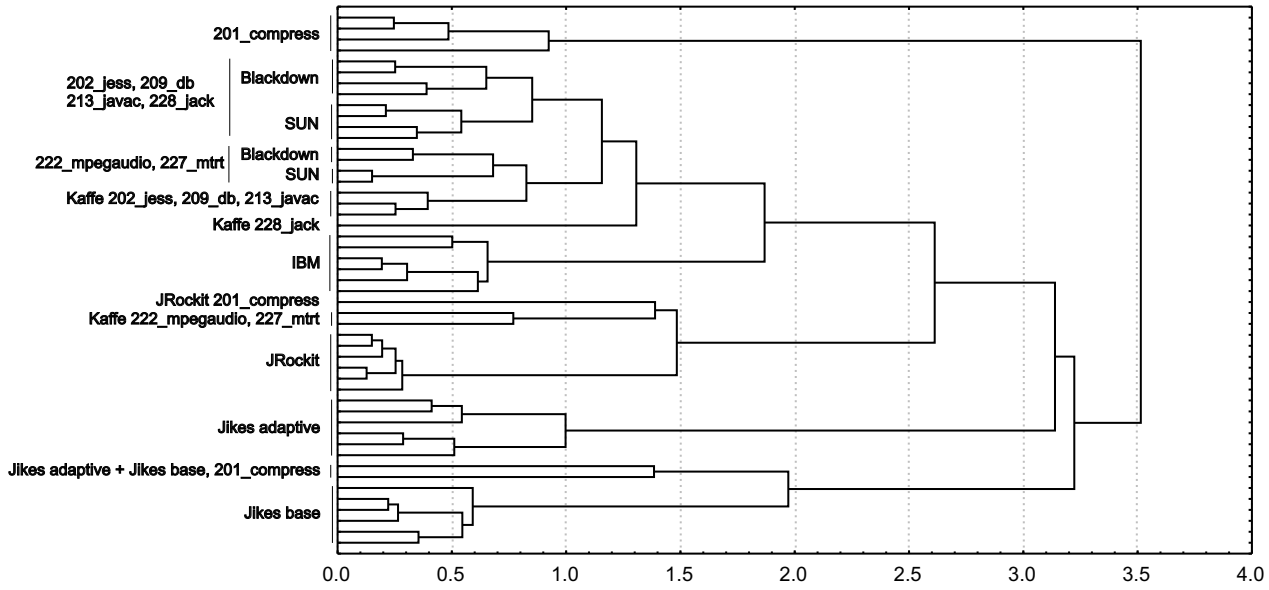

Figure 4: Dendrogram for the SPECjvm98 s1 workload set obtained after cluster analysis using the average pair-group strategy.

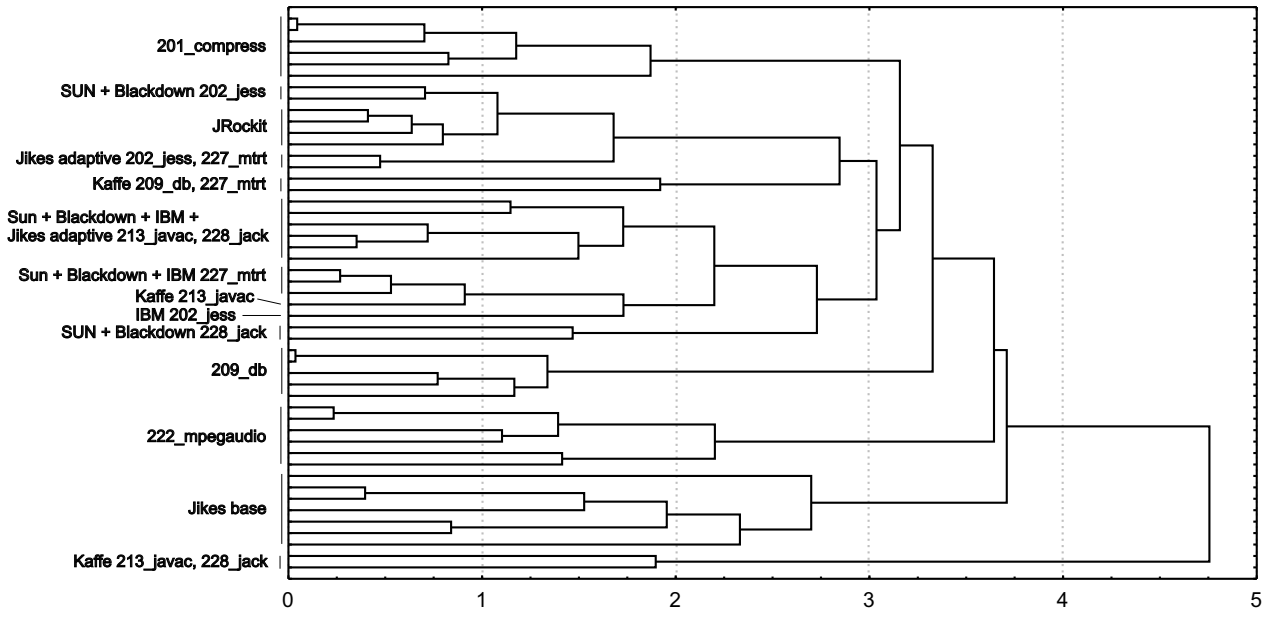

Figure 5: Dendrogram for the SPECjvm98 s100 workload set obtained after cluster analysis using the average pair-group strategy.

The component in negatively influenced, mainly by the number of retired taken branches, the number of retired near returns, and the number of RAS hits.

Although the reduced 6-dimensional space obtained after PCA is significantly smaller than the original 34-dimensional space, displaying a 6-dimensional space in an understandable way is impractical, if not impossible. Therefore, we only display the dendrogram obtained after CA and not the Java workload space as a function of its principal components. This dendrogram is shown in Figure 5. A first interesting observation that can be made from this figure is that the clusters that are formed for the s100 input set are not the same as for s1, compare Figure 5 to Figure 4. Moreover, the clusters that are formed for s100 are not necessarily formed around virtual machines as it was the case for the s1 input set.

For the s100 input set, we observe *benchmark clusters*—the same benchmark being run on different VMs, or small impact of VM on overall behavior—as well as *virtual machine clusters*—the same virtual machine running different Java applications, or large impact of VM on overall behavior. In Figure 5, *we observe three tight benchmark clusters: (i) a cluster corresponding to 201_compress, (ii) a cluster corresponding to 222_mpegaudio, and (iii) a cluster corresponding to 209_db.* The first two clusters contain all the virtual machines except for the baseline version of Jikes. The last cluster around 209_db contains five virtual machines, all but Kaffe and the baseline version of Jikes. Interestingly, Shuf *et al.* [23] labeled these SPECjvm98 benchmarks as 'simple' benchmarks. The fact that the virtual machines running these 'simple' benchmarks result in clustered data points is probably (and surprisingly) due to the fact that all the virtual machines have optimized these simple benchmarks to nearly the same native code during the long-running time of these benchmarks. Note that in contrast to the widespread behavior of 201_compress for the s1 input, the s100 input results in a tight cluster.

In addition to these three 'benchmark clusters', *we observe two tight virtual machine clusters: (iv) the baseline version of the Jikes virtual machine, and (v) the JRockit virtual machine.* The cluster around the baseline Jikes VM contains all the SPECjvm98 benchmarks. The fact that the various Java programs that are run on baseline Jikes exhibit similar behavior can be explained as follows. The baseline configuration of Jikes compiles each method just-in-time but the number of (dynamic) optimizations performed is limited. As such, we can expect that more or less the same code sequences will be generated for different Java programs yielding similar behavior. The cluster around JRockit contains all the SPECjvm98 benchmarks except for 201_compress, 209_db and 222_mpegaudio. Interestingly, these benchmarks are part of the 'benchmark clusters' (i), (ii) and (iii).

From a close inspection of the results in Figure 5, *we also observed that the SUN 1.4.1 VM and the Blackdown 1.4.1 VM yield similar behavior.* Note however, in contrast to the results of s1, that this is only true on a per benchmark basis.

4.2 Java Grande Forum

For the Java Grande Forum (JGF) benchmark suite, which includes four benchmarks each having two problem sizes see also Table 2, we retain six principal components during PCA. These six principal components explain 82.5% of the total variance. The dendrogram obtained from cluster analysis on this 6-dimensional space is shown in Figure 6. From this figure, we can conclude that (i) the Java workloads associated with Kaffe as well as the Java workloads associated with the baseline configuration of Jikes form tight clusters, respectively; (ii) a tight cluster is observed for search: all the virtual machines running search are in the same cluster except for Kaffe and the baseline version of Jikes; (iii) the SUN 1.4.1 VM and the Blackdown 1.4.1 VM also show similar behavior per benchmark, e.g., both virtual machines are close to each other for the euler benchmark; (iv) the small and large problem sizes generally yield the same behavior except for moldyn.

4.3 All the Java workloads

For the analysis discussed in this section, as much as 227 Java workloads are included by varying the virtual machine, the Java application and their input sets. Next to the virtual machine configurations mentioned in Table 1, we added the server mode of the SUN 1.4.1 VM as well as the server mode of the Blackdown 1.4.1 VM. Based on the results of the principal components analysis we retain seven principal components accounting for 82.2% of the total variance. The dendrogram obtained from the cluster analysis done on this 7-dimensional space is shown in Figure 7. Interesting observations can be made from this figure.

- First, a number of virtual machine clusters are observed that contain various Java applications on the same virtual machine, (i) the IBM 1.4.1 VM running the Raja benchmark, (ii) the Jikes baseline configuration, (iii) Kaffe, running several Java Grande Forum benchmarks and some SPECjvm98 benchmarks for the s100 input set, (iv) the adaptive configuration of Jikes, and (v) minor clusters for JRockit and the IBM VM. Note that the SUN 1.4.1 VM and the Blackdown 1.4.1 VM form a single cluster for the Raja benchmark as well as for SPECjbb2000, indicating strong similarities in the behavior of both virtual machines for these benchmarks. For SPECjbb2000, although the client and server modes of the SUN and Blackdown virtual machines are quite close to each other in the global picture (linkage distance smaller than 1.2), we can observe a clear distinction between both. In addition, we also noticed that for SPECjbb2000, the server mode Blackdown 1.4.1 VM shows more similarities with the IBM 1.4.1 VM than with the server mode SUN 1.4.1 VM.

- Second, we observe a number of benchmark clusters containing various virtual machines running the same Java benchmark, e.g, the Java Grande Forum benchmarks (search, moldyn, euler and raytracer), SPECjvm98's 201_compress, SPECjvm98's 209_db with the s100 input and SPECjbb2000.

- Third, we observe two clusters formed around several of the SPECjvm98 benchmarks with the s1 input set, showing once more that these workloads exhibit dissimilar behavior from the other Java workloads.

How these results should be interpreted and used by researchers in the object oriented programming community depends on their research goals. *Virtual machine developers benchmarking their own virtual machine should select*

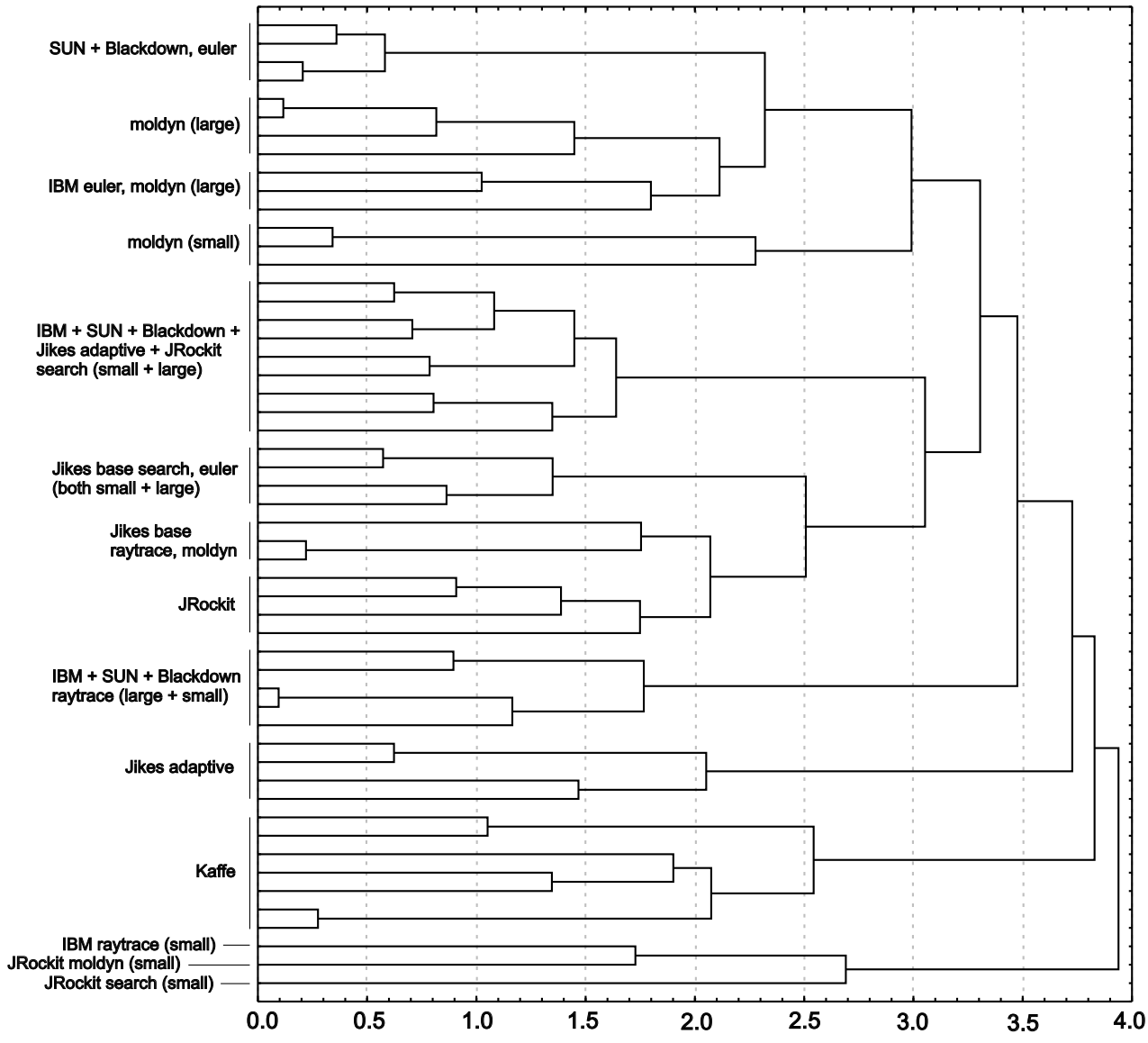

Figure 6: Dendrogram for the Java Grande Forum benchmarks obtained after cluster analysis using the average pair-group strategy.

Figure 7: Dendrogram for all the Java workloads obtained after cluster analysis using the average pair-group strategy.

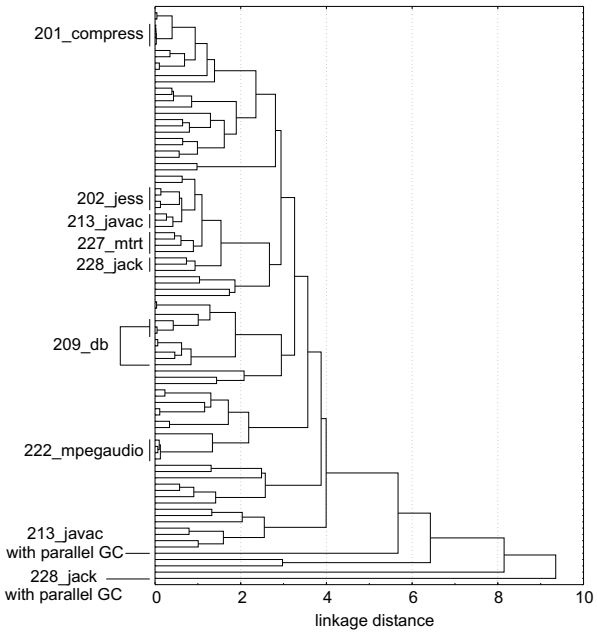

Figure 8: Measuring the impact of the garbage collector on Java workload behavior.

a number of benchmarks that cover a sufficiently large behavioral spectrum for their virtual machine. The collection of benchmarks will thus be different for different virtual machines. For example, for JRockit we recommend SPECjbb2000, 201_compress, 222_mpegaudio, 228_jack, 213_javac, 209_db and the four JGF benchmarks. For the baseline configuration of Jikes on the other hand, we recommend only two SPECjvm98 benchmarks and one JGF benchmark. *Java application developers benchmarking their own Java program are recommended to use a sufficiently large number of virtual machines.* However, our results suggest that it is a waste of effort to consider the SUN VM as well as the Blackdown VM.

4.4 Comments on the garbage collector

As noted in section 2.1.1, the choice of the garbage collector was not consistent, i.e., different virtual machine configurations have different garbage collectors. This was due to the fact that we have chosen the default garbage collector for each virtual machine. To quantify the impact of the choice of the garbage collector on the overall results of this paper, we have set up the following experiment. We considered the SPECjvm98 benchmarks with the s100 input set for the various virtual machine configurations in Table 1. For the JRockit VM we considered three additional garbage collectors next to the generational copying garbage collector, namely single spaced concurrent, generational concurrent and parallel garbage collection. The dendrogram that is obtained after PCA and CA is shown in Figure 8. The four JRockit garbage collectors are highlighted for each SPECjvm98 benchmark. This graph shows that for most benchmarks the various garbage collectors are tightly clustered, except for the parallel garbage collector for 213_javac and 228_jack. As such, we conclude that the choice of the garbage collector in this paper has a minor influence on the overall conclusions of this paper.

5. RELATED WORK

This section discusses related work on understanding and characterizing Java workloads.

Bowers and Kaeli [8] characterize the SPECjvm98 benchmarks at the bytecode level. They conclude that Java applications have a large number of loads in their dynamic bytecode stream.

Hsieh *et al.* [15] compare the performance of the SUN JDK 1.0.2 Java interpreter, a bytecode to native code translator called Caffeine [16] and a compiled C/C++ version of the code. This is done based on simulations. They conclude that the interpreter exhibits poor branch target buffer (BTB) performance, poor I-cache behavior and poor D-cache behavior compared to the other approaches.

Chow *et al.* [10] compare Java workloads with non-Java workloads (e.g., SPEC CPU95, SPEC CINT95, etc.) using principal components analysis. In this study, the authors focus on the branch behavior, i.e., the number of conditional jumps, direct calls, indirect calls, indirect jumps, returns, etc. Based on simulation results, they conclude that Java workloads appear to have more indirect branches than non-Java workloads. However, the number of indirect branch targets can be small. I.e., when considering the number of indirect target changes, Java workloads are no worse than some SPEC CINT95 benchmarks. The study presented in this paper is different from the work done by Chow *et al.* for three reasons. First, although Chow *et al.* use a large number of workloads, the number of virtual machines used in their study is limited to two. Second, Chow *et al.* limit their study to the branching characteristics of Java workloads. Third, the goal of the paper by Chow *et al.* was the compare Java workloads versus non-Java workloads which is different from the goal of this paper, namely getting insight in the interaction between VMs, Java programs and their inputs.

Radhakrishnan *et al.* [20, 21] analyze the behavior of the SPECjvm98 benchmarks by instrumenting the virtual machines and by simulating execution traces. They used two virtual machines: the Sun JDK 1.1.6 and Kaffe 0.9.2. They conclude that (i) 45 out of the 255 bytecodes constitute 90% of the dynamic bytecode stream, (ii) an oracle translation scheme (optimal translation selection) in case of a JIT compiler can only improve performance by 10% to 15%, (iii) the I-cache and D-cache performance is better for Java applications than for C/C++ applications, except for the D-cache in JIT mode, (iv) write misses due to installing JIT compiler output have a significant impact on the D-cache performance in JIT mode, and (v) the amount of ILP is higher under JIT mode than under interpreter mode.

Li *et al.* [19] characterize the behavior of SPECjvm98 Java benchmarks through complete system simulation. This was done by using the Sun JDK 1.1.2 virtual machine and the SimOS complete system simulator [22]. They conclude that the SPECjvm98 applications (on s100) spend on average 10% of their time in system (kernel) activity compared to only 2% for the four SPEC CINT95 benchmarks studied. Generally, the amount of time in kernel activity is higher for the JIT compiler mode than for the interpreter mode. The kernel activity is mainly due to TLB miss handler invocations. Also, they conclude that the SPECjvm98 benchmarks have inherently poor instruction-level parallelism (ILP) com-

pared to other classes of benchmarks.

In [18], Li *et al.* analyze the impact of kernel activity on the branch behavior of Java workloads. They conclude that branches in OS code exhibit a different biased behavior which increases the branch misprediction rate significantly. As such, they propose OS-aware branch prediction schemes which outperform conventional branch predictors.

Shuf *et al.* [23] characterize the memory behavior of Java workloads. They conclude that some SPECjvm98 benchmarks are not truly object-oriented and are thus not representative for real Java workloads. As such, they propose to use the server-oriented pBOB benchmark [5] in studies on Java workloads in addition to some SPECjvm98 benchmarks. In our experiments, we used the SPECjbb2000 benchmark suite which is based on pBOB. The results presented in this paper indeed confirm that the behavior that is observed for SPECjbb2000 is dissimilar from SPECjvm98. Secondly, they conclude that the number of hot spots is small for most Java programs. Consequently, expensive algorithms are justified for run-time optimizations. Third, they conclude that the D-cache behavior of Java workloads is poor resulting in high D-cache miss rates—even fairly large L2 caches do not increase performance significantly. In addition, they conclude that the TLB as well as the cache behavior is worse for Java workloads than for technical benchmarks, but comparable to commercial workloads.

6. CONCLUSIONS

This paper studied how much of the behavior of a Java workload as seen at the microarchitectural level is due to the virtual machine, the Java application itself and the input to the Java application. In other words, we addressed the question whether the behavior of a Java workload is primarily determined by the virtual machine, the Java application or its input. In the experimental setup of this paper, we used seven virtual machine configurations and a collection of Java benchmarks taken from SPECjvm98 (with varying input sets `s1`, `s10` and `s100`), SPECjbb2000, the Java Grande Forum as well as an open-source raytracer called Raja with a large number of scene descriptions. For each of these workloads, a number of performance characteristics were measured through hardware performance counters on an AMD K7 Duron microprocessor. This large amount of data was subsequently analyzed using two statistical data analysis techniques, namely principal components analysis and cluster analysis. These data reduction techniques gave us an excellent opportunity to answer the questions raised in this paper.

From this paper, we conclude that:

- *for the* `s1` *input set of SPECjvm98, the behavior as observed at the microarchitectural level is mainly determined by the virtual machine.* This is due to the fact that the `s1` input set leads to short-running benchmarks. This causes the startup of the virtual machine to be the largest contributor to the overall behavior. As such, *this suggests that using the* `s1` *input set in a Java system performance analysis might not be good practice (unless one is mainly interested in measuring startup time)* since the results that are obtained from such an analysis can be highly biased by the virtual machine that is used.

- *using the short-running* `s1` *input set as a representative for the long-running* `s100` *input set of SPECjvm98 is clearly not good practice*, since the behavior that is observed at the microarchitectural level can be quite different for both input sets. One reason obviously is the fact that a virtual machine has more opportunities for run-time optimizations for long-running benchmarks than for short-running benchmarks.

- *for the Java Grande Forum benchmark suite on the other hand, the problem size seems to have a minor impact on the overall behavior in most cases.* As such, the smallest problem size can be used with confidence.

- *for the SPECjvm98* `s100` *input set, 'virtual machine clusters' are observed containing various virtual machines running the same Java program as well as 'benchmark clusters' containing various Java benchmarks running on the same virtual machine.* This implies that for the 'virtual machine clusters' the impact of the Java application is higher than the impact of the virtual machine. Interestingly, these virtual machine clusters are observed for previously reported 'simple' benchmarks, namely 201_compress, 209_db and 222_mpegaudio. Analogeously, for the 'benchmark clusters' the impact of the virtual machine is higher than the impact of the Java program. An example of a 'benchmark cluster' is the baseline configuration of the Jikes virtual machine.

- *for the SPECjbb2000 benchmark run on aggressive runtime optimizing virtual machines, we observe a behavior that is very dissimilar to other Java workloads.* As such, including a server-oriented Java workload is important to obtain a representative Java workload.

- in general, researchers should be careful when reporting results using only one or two virtual machines. *The results presented in this paper clearly show that the behavior that is observed at the microarchitectural level is highly dependent on the virtual machine.* As such, results obtained for one virtual machine might not be applicable for another virtual machine and vice versa.

Again, we want to emphasize the importance of the results and the conclusions presented in this paper for the object oriented programming community. This paper clearly showed that the selection of representative Java workloads can be done based on scientific arguments. Indeed, principal components analysis and cluster analysis provide researchers valuable information to reason about the quality of their Java workloads in a reliable way. This will allow them to draw conclusions from their studies with more confidence.

Acknowledgements

Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research – Flanders (Belgium) (F.W.O. Vlaanderen). Andy Georges is supported by the SEESCOA project sponsored by the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT – Vlaanderen). The authors would also like to thank the anonymous reviewers for their valuable comments.

7. REFERENCES

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

[2] 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.

[3] M. Arnold, S. Finka, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, pages 47–65, October 2000.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[5] S. J. Baylor, M. Devarakonda, S. J. Fink, E. Gluzberg, M. Kalantar, P. Muttineni, E. Barsness, R. Arora, R. Dimpsey, and S. J. Munroe. Java server benchmarks. *IBM Systems Journal*, 39(1):57–81, February 2000.

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

[7] P. Bose. Performance evaluation and validation of microprocessors. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99)*, pages 226–227, May 1999.

[8] K. R. Bowers and D. Kaeli. Characterizing the SPECjvm98 benchmarks on the Java Virtual Machine. Technical Report ECE-CEG-98-026, Northeastern University, Department of Electrical and Computer Engineering, September 1998.

[9] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency, Practice and Experience*, 12(6):375–388, May 2000.

[10] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proceedings of the Workshop on Workload Characterization (WWC-1998), held in conjunction with the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-31)*, pages 11–19, November 1998.

[11] R. Desikan, D. Burger, and S. W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA-28)*, pages 266–277, July 2001.

[12] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(14), October 1998.

[13] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Designing workloads for computer architecture research. *IEEE Computer*, 36(2):65–71, February 2003.

[14] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, February 2003. `http://www.jilp.org/vol5`.

[15] C. A. Hsieh, M. T. Conte, T. L. Johnson, J. C. Gyllenhaal, and W. W. Hwu. A study of the cache and branch performance issues with running Java on current hardware platforms. In *Proceedings of the COMPCON'97*, pages 211–216, February 1997.

[16] C. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 90–99, December 1996.

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

[18] T. Li, L. K. John, A. Sivasubramaniam, N. Vijaykrishnan, and J. Rubio. Understanding and improving operating system effects in control flow prediction. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 68–80, October 2002.

[19] T. Li, L. K. John, N. Vijaykrishnan, A. Sivasubramaniam, J. Sabarinathan, and A. Murthy. Using complete system simulation to characterize SPECjvm98 benchmarks. In *Proceedings of the 14th International Conference on Supercomputing (ICS-2000)*, pages 22–33, May 2000.

[20] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, and A. Sivasubramaniam. Architectural issues in java runtime systems. In *Proceedings of the Sixth International Symposium on High Performance Computer Architecture (HPCA-6)*, pages 387–398, January 2000.

[21] R. Radhakrishnan, N. Vijaykrishnan, L. K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Transactions on Computers*, 50(2):131–145, February 2001.

[22] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(1):78–103, January 1997.

[23] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the memory behavior of java workloads: a structured view and opportunities for optimizations. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, June 2001.

[24] StatSoft, Inc. STATISTICA for Windows. Computer program manual. 1999. `http://www.statsoft.com`, 1999.

[25] 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.

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