A Multifaceted Memory Analysis of Java Benchmarks

Orion Papadakis
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Andreas Andronikakis
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Nikos Foutris
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Michail Papadimitriou
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Athanasios Stratikopoulos
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Foivos S. Zakkak
Red Hat Inc.
Manchester, United Kingdom
fzakkak@redhat.com

Polychronis Xekalakis
Nvidia
Portland, USA
pxekalakis@nvidia.com

Christos Kotselidis
The University of Manchester
Manchester, United Kingdom
{first}.{last}@manchester.ac.uk

Abstract

Java benchmarking suites like Dacapo and Renaissance are employed by the research community to evaluate the performance of novel features in managed runtime systems. These suites encompass various applications with diverse behaviors in order to stress test different subsystems of a managed runtime. Therefore, understanding and characterizing the behavior of these benchmarks is important when trying to interpret experimental results.

This paper presents an in-depth study of the memory behavior of 30 Dacapo and Renaissance applications. To realize the study, a characterization methodology based on a two-faceted profiling process of the Java applications is employed. The two-faceted profiling offers comprehensive insights into the memory behavior of Java applications, as it is composed of high-level and low-level metrics obtained through a Java object profiler (NUMAProfiler) and a microarchitectural event profiler (PerfUtil) of MaxineVM, respectively. By using this profiling methodology we classify the Dacapo and Renaissance applications regarding their intensity in object allocations, object accesses, LLC, and main memory pressure. In addition, several other aspects such as the JVM impact on the memory behavior of the application are discussed.

CCS Concepts: • **Software and its engineering → Memory management**; **Runtime environments**; **Object oriented languages**.

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622978

Keywords: JVM, Memory Profiling, Managed Runtimes, Dacapo, Renaissance, MaxineVM

ACM Reference Format:
Orion Papadakis, Andreas Andronikakis, Nikos Foutris, Michail Papadimitriou, Athanasios Stratikopoulos, Foivos S. Zakkak, Polychronis Xekalakis, and Christos Kotselidis. 2023. A Multifaceted Memory Analysis of Java Benchmarks. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617651.3622978

1 Introduction

The in-depth understanding of the memory behavior of standardized benchmarking suites, such as the traditional Dacapo [3], as well as the later Renaissance [23], is essential for the community of JVM researchers and practitioners. That said, memory profiling for Java applications is a challenging task due to the "noise" introduced by the JVM itself. The JVM interference lowers the accuracy of coarse-grain, black-box profiling[1], while on the other hand, fine-grain wrapping of the application code imposes technical challenges as it is intrusive and requires source code recompilation.

Popular Java profilers offer a range of high-level metrics, including object allocations, threads, GC, and more. However, such tools typically provide only the high-level profile of a Java application, lacking correlation with the underlying hardware, and thus making the analysis susceptible to blind spots and inconsistent conclusions. For instance, the Dacapo *sunflow* is characterized by [15] as "memory-intensive" with respect to the total object allocations and the allocation rate. However, in this paper we discover that a close inspection of the last level cache misses reveals that the

[1]We refer to black-box profiling the process of using external tools to profile the VM as whole without distinguishing any execution phases or VM-internal aspects such as mutator or GC threads.

application does not pose high pressure to the main memory due to its good data locality. Hence, although sunflow indeed is memory intensive with respect to the number of total allocations, as we also observe, there is no negative impact to its performance. Such a case highlights the value of co-examining low-level hardware-related metrics alongside typical high-level metrics to avoid misconceptions when profiling managed applications. Similarly, approaches that solely focus on low-level metrics lack the reverse correlation and consequently turn out to be insufficient for the same reasons. For the aforementioned reasons, a *multifaceted* characterization approach that combines metrics across different layers of the stack is needed.

This paper, addresses this gap by proposing a methodology to characterize the memory behavior of a Java application by analyzing and correlating application and hardware profiling metrics. The proposed methodology is multifaceted as it employs two independent profilers of MaxineVM [13]: NUMAProfiler [21], and PerfUtil [21, 22]. The former profiler monitors the VM to collect high-level metrics, such as object allocations, accesses, etc. This information enables an initial characterization of the memory intensity of an application. The latter profiler leverages the Hardware Performance Counters of the system to collect low-level metrics related to hardware. This information is helpful to confirm or reconsider the outcome of the initial characterization. The co-examination of high and low-level metrics reduces potential profiling blind-spots and provides valuable insights for both the extensively studied Dacapo benchmarks and the newer Renaissance benchmark suite. Hence, this paper makes the following contributions:

1. It proposes a methodology to effectively characterize the memory behavior of a Java application based on a multifaceted profile that is composed of high-level and low-level metrics.
2. It presents a comprehensive study on the memory behavior of 30 Dacapo and Renaissance benchmarks. The study not only showcases the effectiveness of the proposed methodology through the discussion of selected examples but also results in the classification of the studied benchmarks into several categories, leveraging the multifaceted profile.

The rest of the paper is organized, as follows. Section 2 presents the tools that are utilized to perform the multifaceted profiling. Section 3 describes the experimental methodology that is followed for profiling all benchmarks with the selected profilers as well as the experimental testbed. Section 4 presents a rigorous study on the memory behavior of all benchmarks. In particular, Section 4.2 performs an initial characterization of the applications using the application-level metrics obtained by NUMAProfiler, whereas Section 4.3 expands the initial characterization while discussing the microarchitectural metrics obtained by PerfUtil. Finally, Section 5 presents the related work, and Section 6 conveys the conclusions.

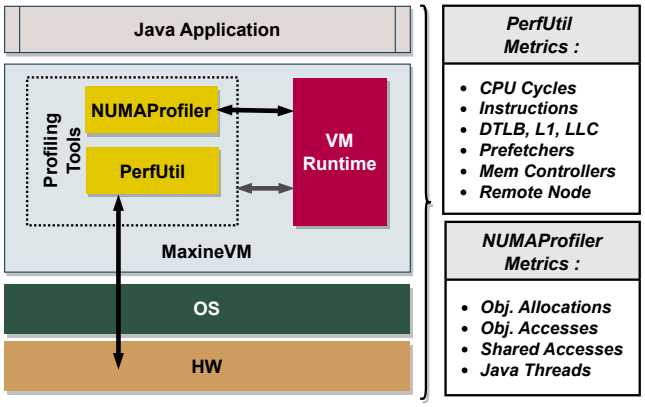

Figure 1. MaxineVM equipped with PerfUtil & NUMAProfiler.

2 Tooling support for multifaceted profiling

This section presents the two profiling tools that operate within MaxineVM, a metacircular research VM written in Java. NUMAProfiler is employed to collect the high-level metrics related to the application layer, while PerfUtil is used to obtain low-level microarchitectural metrics. Both profilers have been independently validated against other functionality equivalent ones [20]. Even though MaxineVM and its profiling tools do not suggest a production environment, this paper stands as an effective proof-of-concept and aims to point towards new profiling opportunities for managed runtimes. Figure 1 illustrates the software stack of MaxineVM along with an overview of the data metrics that are being collected by each profiler. More information about each profiler is given in the following sections.

2.1 NUMAProfiler

NUMAProfiler is an accurate Java object profiler for MaxineVM that is also enriched with NUMA awareness [21]. It probes the runtime layer of the VM in order to monitor object allocations, object accesses, survivor objects after garbage collection, threads as well as the NUMA placement of the virtual pages in the heap. NUMAProfiler exposes an API to the VM runtime. The API calls are injected into the proper components of MaxineVM. To avoid heap pollution as well as the interruption of the application's threads, NUMAProfiler maintains thread-local buffers off-heap to store the profiling data. Additionally, the API calls are used to lazily trigger the profiler mechanisms when it is necessary. Even though the profiling process is simplified due to this lazy approach,

substantial overhead is introduced by the profiler ($\sim 10x$). However, that overhead can be tolerated since NUMAProfiler is intended for offline profiling purposes. While the NUMA-related features of the profiler pertain to NUMA hardware and its implications, this paper focuses solely on studying the memory behavior of Java applications within a traditional CPU architecture with uniform memory access. Hence, NUMA architecture which is orthogonal to the current study, is not within the scope of this work.

A feature of NUMAProfiler is the classification of the *ownership* for the object accesses. NUMAProfiler classifies an object access as *shared* or *thread-local*. Such a classification is an important application property [12] because it highlights the inter-thread dependencies; nevertheless, it is not a trivial task. An object access is considered as shared, if the thread that performed the access and the thread that acts as the object owner are different. MaxineVM is modified to store the owner of each object into the misc word of the object header. Hence, the owner thread is disclosed during the profiling of an object access along with the thread that performs the access. For this work, NUMAProfiler is tuned to consider that the owner of an object is the thread that has allocated this object (allocator thread).

2.2 PerfUtil

PerfUtil is an accurate and flexible profiler which equips the VM itself with fine-grain utilization of the Hardware Performance Counters [22]. It interfaces with the *perf* [9] functionality of the Linux kernel, and it passes over the control to the Java code of the VM. PerfUtil offers a flexible and customizable way of monitoring microarchitectural metrics per thread, per core or both. Similarly to NUMAProfiler, the functionality of PerfUtil is exposed to the VM runtime by an API. In addition, PerfUtil supports time-multiplexing that enables a large set of events to be simultaneously counted, while it operates with low overhead [21].

3 Experimental Setup

3.1 Testbed Characteristics

Table 1 shows the hardware and software characteristics of the testbed that we used. The testbed is a Dell PowerEdge R620 server that contains a dual socket Intel Xeon processor with two NUMA nodes that result in 32 number of cores. To ensure that any NUMA-related effect that might influence performance is excluded when studying the memory behavior of the selected benchmarks, we employ the *Single Node* configuration (see Table 1). That configuration establishes a Uniform Memory Access (UMA) environment for performing our experiments by utilizing only one NUMA node. Moreover, the Intel hyper-threading technology is disabled to prevent any additional variations with regard to the performance or the memory behavior of the benchmarks. To avoid dynamic voltage and frequency scaling (DVFS), the CPU frequency is fixed to at 2.9 GHz via the ACPI CPU frequency driver.

Table 1. Hardware and Software Configurations.

HW	Processor	2 x Intel Xeon E5-2690
	Sockets	2
	NUMA nodes	2
	Num of Cores	16 (32 threads)
	LLC Size	40MB
	Memory Controllers	8
	DRAM	384GB
SW	OS	Ubuntu 16.04
	Kernel	Linux 4.15.0-112-generic
	JVM	MaxineVM 2.9
	GC	SemiSpace (non generational)
		Single Node
# of CPUs		1
# of Utilized Cores		8
LLC Size (MB)		20
Mem. Controllers		4
DRAM Size (GB)		192
Java Heap Size (GB)		100
HyperThreading		off
Page migration		off

3.2 Benchmark Suites

The latest pre-built maintenance release of Dacapo benchmarks (dacapo 9.12 MR1) [4] was used, while the pre-built 0.11.0 release[2] was used for Renaissance. The number of iterations of the benchmarks was selected based on the well known good practices [15, 23] to reach a warmed up state and it was augmented by ten additional runs to include enough run-steady iterations in our measurements. Moreover, Dacapo allows the user to configure the input size and the deployed threads with some exceptions (i.e., `avrora`) where the number of threads is determined by the input size. The benchmarks of Renaissance have a "test" (small) and a "jmh" (default/large) input size and most of the benchmarks aim to automatically deploy worker threads equal to the number of available cores. In our experiments, we used the largest input size and deployed eight threads (wherever possible) which corresponds to the number of cores in a single node. Table 2 lists the studied Dacapo and Renaissance applications, along with their run configurations. Note that some applications (`batik`, `eclipse`, `tomcat`, `tradebeans`, `tradesoap`, `dec-tree`, `finagle-chirper`, `finagle-http`, `page-rank`) are omitted from the performance evaluation due to various failures of MaxineVM including memory corruption (segfaults) or concurrency bugs that lead to livelocks.

3.3 Experimental Methodology

The experiments were conducted in a two-step process. Each step corresponds to an individual build of MaxineVM equipped exclusively with one of the two profilers to prevent interference between the profilers that may skew the results. The first step, deploys MaxineVM with NUMAProfiler to

[2] https://github.com/renaissance-benchmarks/renaissance/tree/v0.11.0

Table 2. Dacapo & Renaissance Benchmarks Characteristics.

	Benchmark	Input Size	Iterations	Instructions	L1D Reads	L1D Writes	Total Mem.	Arith.	Branch
Dacapo	avrora	large (max)	30	57,645,363,283	35%	18%	56%	29%	15%
	fop	default (max)	50	2,644,295,397	28%	25%	47%	38%	15%
	h2	huge (max)	20	1,292,129,013,772	29%	16%	48%	37%	15%
	jython	large (max)	30	294,432,673,937	27%	32%	43%	40%	16%
	luindex	default (max)	50	5,085,147,760	29%	25%	43%	40%	18%
	lusearch	large (max)	30	72,686,381,316	28%	21%	43%	40%	17%
	lusearch-fix	large (max)	30	72,517,120,274	28%	21%	43%	40%	17%
	pmd	large (max)	30	36,072,750,432	28%	22%	46%	38%	16%
	sunflow	large (max)	30	162,257,371,836	31%	21%	43%	43%	15%
	xalan	large (max)	30	223,669,834,459	29%	21%	48%	36%	15%
	GEOMEAN - D	-	-	67,741,600,233	29%	22%	46%	38%	16%
Renaissance	akka-uct	N/A	34	413,031,096,621	30%	16%	45%	36%	19%
	reactors	N/A	20	344,901,909,214	31%	21%	52%	33%	15%
	als	N/A	40	173,577,079,497	24%	9%	33%	50%	16%
	chi-square	N/A	70	44,712,182,871	25%	15%	40%	42%	18%
	gauss-mix	N/A	50	88,927,702,969	26%	16%	42%	40%	18%
	log-regression	N/A	30	66,506,869,326	35%	8%	44%	38%	18%
	movie-lens	N/A	30	255,423,610,313	25%	14%	39%	45%	16%
	naive-bayes	N/A	40	95,767,198,846	28%	9%	37%	45%	18%
	db-shootout	N/A	26	415,462,411,962	26%	17%	43%	41%	16%
	fj-kmeans	N/A	40	16,587,430,766	33%	14%	46%	38%	15%
	future-genetic	N/A	60	50,065,010,991	35%	23%	58%	30%	13%
	mnemonics	N/A	26	167,497,623,059	27%	19%	46%	38%	16%
	par-mnemonics	N/A	26	164,897,514,508	27%	19%	46%	38%	16%
	scrabble	N/A	60	44,953,086,743	28%	19%	48%	37%	15%
	neo4j-analytics	N/A	30	176,983,336,923	29%	16%	44%	39%	16%
	rx-scrabble	N/A	90	6,552,607,394	32%	25%	56%	31%	13%
	dotty	N/A	50	19,001,592,728	26%	16%	42%	41%	17%
	scala-doku	N/A	20	48,122,342,431	30%	12%	42%	41%	17%
	scala-kmeans	N/A	60	9,360,648,141	31%	18%	48%	38%	13%
	philosophers	N/A	40	90,682,695,138	31%	20%	51%	34%	15%
	scala-stm-bench7	N/A	70	21,856,846,101	28%	19%	47%	38%	15%
	GEOMEAN - R	-	-	72,999,411,837	29%	16%	45%	39%	16%
	GEOMEAN	-	-	71,260,215,626	29%	17%	45%	38%	16%

collect various object-related metrics, while the second step runs MaxineVM with PerfUtil to collect numerous microarchitectural metrics. Note that the non-determinism of Dacapo and Renaissance was experimentally observed to have minimal impact on our results. We verified this by comparing and contrasting the two runs as follows. First, we compared the instruction count between runs and the total cycles required to complete the runs. Then, we compared the cache behavior of benchmarks across the two runs ensuring the miss ratios are similar. Naturally, the two runs have slightly different absolute numbers; however, the behavior of the benchmarks was almost identical. To reach parity between the two runs we ensured that all timed executions were in hot state (i.e. almost no recompilation was taking place). In addition, we ran all experiments with the same configurations and with large heap sizes to ensure minimal interference from the GC. Furthermore, when comparing against OpenJDK runs (Section 4.6), we configured OpenJDK to behave as similar as possible to MaxineVM by de-activating optimizations that are not present in MaxineVM (e.g. escape analysis, compressed pointers, etc.). Finally, the multiplexing feature of PerfUtil enabled concurrent monitoring of thirty-two perf events in a single run.

4 Study

4.1 Methodology

Performing such a multifaceted performance analysis - involving two different profilers - produces a large number of data points which may be difficult to navigate and draw conclusions. Below we provide a methodology, based on our experience, on how to interpret those numbers in order to characterize various benchmarks.

In general, we can follow two approaches: bottom-up or top-down. In the bottom-up approach we start by looking into the micro-architectural characterization of a benchmark trying to understand what factors affects its performance.

Then, we move at a higher level by comparing and contrasting the numbers achieved by the high-level profiler in order to validate or complement our assumptions and understanding based on the low-level metrics. By examining the results from the low-level profiler in Table 4, the first metric we focus on is that of the CPI (Cycles Per Instruction). In general, the larger the CPI the slower the benchmark is. If the CPI is high, we typically check the three main factors that affect performance on modern processors: branch misprediction ratio, cache miss ratio, and TLB miss ratio.

Based on the observed numbers, we hypothesize about the behavior of the benchmark and then we try to the validate those hypotheses by comparing and contrasting the low-level results with those from the high level profiler. For example, if we notice that a benchmark has high cache miss ratios, which correlates with high CPI, we examine its allocation rate and size to determine whether this is the root cause of the problem or whether the benchmark has just irregular memory access patterns. The same logic can be applied for other metrics.

In the top-down approach, we follow a reverse methodology. We first look into the high-level performance metrics of NUMAProfiler to get a high-level understanding of the benchmark and then we start delving deeper into its performance characteristics. By examining first the high-level performance metrics we can identify potential performance bottlenecks of a benchmark and then focus on the low-level microarchitectural profiling results that regard these specific potential bottlenecks (e.g., high allocation rates may result to high cache or TLB miss ratios).

In the following subsections, after presenting the collective results from both profilers for all benchmarks, we apply the methodology across two particular benchmarks as a guideline (Section 4.5).

4.2 Characterization With High-Level Metrics

The object allocations, accesses, and their rates over time, are quite indicative regarding the memory intensity of an application. However, they do not always lead to well-rounded conclusions as highlighted in this section. Table 3 (inspired by Lengauer et al. [15]) outlines several object-related metrics (as obtained via NUMAProfiler) for each application. The notable observations and findings of those metrics are discussed in the following subsections. The reported numbers derive from the average of ten run-steady iterations (after warm-up), and the maximum value of each metric is highlighted as bold. Moreover, note that the NUMAProfiler numbers inevitably incorporate MaxineVM-internal objects due to metacircularity. Thus, any observed difference against a HotSpot-based profiler (i.e., AntTracks [14, 15]) is expected and attributed to the effect of metacircularity [20].

4.2.1 Object Allocations Count & Rate.

The total count of object allocations and memory footprint indicate how much memory is allocated per application. However, they do not highlight the intensity of the memory allocation. The object count and object size per second metrics should be taken under consideration towards characterizing the memory intensity of a managed application. An application that allocates new objects at a high rate is very likely to put excessive pressure on the memory system.

Dacapo: `H2` allocates overall the most objects and size of memory, however it has a low allocation rate. This is due to the large number of instructions (and consequently execution time) that `h2` has (see Table 2). `Sunflow` allocates less and smaller objects but it is the most allocation intensive application in terms of objects and memory size. `Jython` is the most intensive single-threaded benchmark both in terms of objects and memory size allocation. `Lusearch-fix` has been introduced as an update to `lusearch` bearing a fix in the Lucene platform that reduces object allocations; however, no such difference is observed[3].

Renaissance: `Akka-uct`, `naive-bayes`, `neo4j-analytics`, `h2`, and `gauss-mix` allocate the most objects in total (per iteration). `Akka-uct` allocates almost double the objects of `naive-bayes` which is the second highest allocating application. `Mnemonics` and `scala-doku` are the single-threaded applications with the most total object allocations. `Naive-bayes`, `akka-uct`, `gauss-mix`, `neo4j-analytics`, `db-shootout`, and `scrabble` are the most intensive in terms of both object and size allocation rate.

4.2.2 Memory Footprint & Object Layout.

The overall object allocations do not necessarily reflect the memory allocation footprint size (per iteration, in MB).

Dacapo: `Luindex` allocates the largest objects on average, and it contains the most and longest arrays. However, it is a single-threaded application with the smallest memory footprint among the Dacapo applications. `Lusearch` and `xalan` follow in terms of average object size showing also higher array rate and average array length than the geometric mean of Dacapos. `Xalan` is an application that has large objects, on average. Even though it performs fewer allocations than `sunflow`, it ends up with a higher memory footprint.

Renaissance: `Fj-kmeans` has by far the largest average object size (1.13 kB) among Renaissance applications, while `log-regression`, `db-shootout`, and `movie-lens` follow. It is notable that although `fj-kmeans` and `db-shootout` allocate fewer objects than `naive-bayes` or `neo4j-analytics`, they end up with higher memory footprint which is apparently related with object size. In addition, `fj-kmeans` is an array-dominated application with 63.9% of its allocations being arrays while `db-shootout`, and `movie-lens` follow.

The discussion above highlights that such metrics are crucial especially when the observed memory footprint origin

[3]This issue has been communicated to the authors of Dacapo, and a fix will be issued in the next release update.

Table 3. Object Allocations, Layout and Accesses in Dacapo & Renaissance.

	Allocations				Object Layout			Accesses			Sh. Accesses	
Application	Objects [M]	Size [MB]	$\frac{MObjects}{sec}$	$\frac{MB}{sec}$	Avg obj size [b]	Array Rate [%]	Avg Array Length	Count [M]	$\frac{MAccesses}{sec}$	R/W Ratio	R [%]	W [%]
avrora	9.3	407	0.4	18	46	22	11	4,865	219	10:1	**93**	**57**
fop	2.8	159	4.7	268	60	32	20	65	109	9:1	0	0
h2	**461.5**	**24,763**	0.9	49	56	34	12	**51,518**	103	16:1	41	31
jython	211.0	16,987	4.6	370	84	28	57	7,518	164	10:1	0	0
luindex	0.2	26	0.2	27	**145**	**39**	**202**	134	139	13:1	0	0
lusearch	30.8	3,876	12.7	1,601	132	25	119	2,357	974	15:1	24	4
lusearch-fix	30.8	3,876	12.7	1,603	132	25	119	2,362	977	14:1	24	5
pmd	27.4	1,541	3.9	219	59	23	32	1,144	162	10:1	36	7
sunflow	175.8	7,344	**43.2**	**1,807**	44	3	3	7,504	**1,847**	**30:1**	84	0
xalan	113.7	13,025	11.3	1,296	120	36	57	7,433	740	12:1	19	23
GEOMEAN-D	**27.5**	**2,085**	**3.7**	**282**	**80**	**23**	**35**	**2,378**	**322**	**13:1**	**12**	**4**
akka-uct	**1,052.2**	**56,561**	69.0	3,711	56	2	9	**23,126**	**1,517**	5:1	69	3
reactors	387.1	15,942	6.6	270	43	18	13	10,496	178	6:1	**77**	**56**
als	62.4	2,671	7.7	328	45	5	146	**1,649**	203	22:1	9	16
chi-square	105.6	3,474	22.6	742	34	2	42	1,000	214	8:1	27	0
gauss-mix	457.5	13,180	61.3	1,766	30	1	43	2,301	308	6:1	36	0
log-regression	11.3	1,068	1.6	148	99	8	545	1,984	274	110:1	2	2
movie-lens	176.0	12,749	7.5	546	76	22	113	4,391	188	10:1	34	11
naive-bayes	561.6	12,917	220.9	5,080	24	0.05	96	1,215	478	112:1	3	2
db-shootout	429.7	36,004	29.5	2,475	88	51	59	4,343	299	3:1	28	6
fj-kmeans	17.1	18,844	1.3	1,449	**1,159**	**64**	225	15,668	1,204	**120:1**	59	1
future-genetic	40.3	2,285	6.2	353	59	4	69	2,053	317	18:1	37	5
mnemonics	260.8	12,709	7.5	366	51	18	17	4,996	144	7:1	0	0
par-mnemonics	261.0	12,724	9.2	448	51	18	17	5,001	176	7:1	47	0
scrabble	66.4	2,982	23.4	1,052	47	15	6	1,411	498	8:1	47	0
neo4j-analytics	524.1	17,751	29.8	1,008	36	4	12	10,151	576	3:1	51	1
rx-scrabble	10.9	452	7.2	299	44	5	9	201	133	6:1	48	0
dotty	48.6	1,726	7.1	252	37	6	66	584	85	13:1	0	0
scala-doku	174.4	4,177	17.2	411	25	1	10	1,393	137	71:1	0	0
scala-kmeans	6.0	195	3.5	114	34	0.11	**675**	373	219	27:1	0	0
philosophers	68.4	4,115	14.0	841	63	15	19	3,802	777	6:1	60	30
scala-stm-bench7	35.0	1,724	11.8	578	52	14	22	894	300	9:1	37	0
GEOMEAN-R	**101.2**	**5,216**	**12.3**	**633**	**54**	**5**	**40**	**2,405**	**292**	**13:1**	**14**	**0.4**
GEOMEAN	**66.4**	**3,880**	**8.3**	**487**	**61**	**8**	**38**	**2,397**	**301**	**13:1**	**16**	**0.1**

matters (i.e., GC optimizations, an optimization targeting large objects, heap size tuning and more). Moreover, the Object Layout metrics reveal additional properties of an application which are very likely to affect its memory and/or overall behavior. For example, large objects (as in `lusearch` and `xalan`) are likely to span across two memory pages. Such applications can stress TLBs and page-tables more than others. This type of memory pressure can be a source of inefficiency in the context of NUMA [1, 10]. For example, `lusearch` and `xalan` have been proven unfriendly to the Page Migration mechanism of Linux [22] which is tightly related to the TLB and page-table. `Lusearch` slows down by ~13% while `xalan` gets its remote node accesses increased by ~300% when Page Migration is enabled.

4.2.3 Object Accesses. The object accesses highlight the application-memory relation degree as observed from the application layer. The columns "Object Accesses" and "Sh. Accesses" of Table 3 present a collection of object access metrics as well as the percentage of shared accesses. The latter refers to the number of accesses performed by a different thread rather than the "owner" of the object (recall Section 2.1) as a percentage of total object accesses.

Dacapo: As can be observed in Table 3, `h2` performs the most object accesses in total while, `sunflow`, `xalan`, and `avrora` have more than 2x more accesses than the geomean. `Sunflow` performs the most object accesses per second followed by `lusearch` and `xalan`. All applications are read-dominated with `sunflow` having the most (30) reads per one write. `Avrora` shows the highest shared object R/W access rate. `Sunflow` follows, but it has only shared Read accesses. A high percentage of shared Read accesses is an indication for the existence of the producer-consumer memory access pattern. Finally, `avrora`, `h2`, and `xalan` show the highest percentage of shared writes.

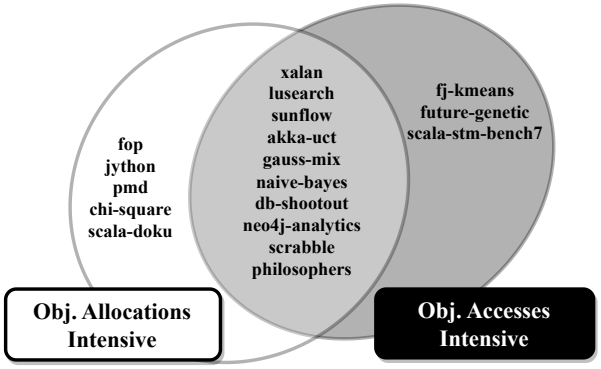

Figure 2. Memory Intensive Applications in terms of Object Allocations and Accesses.

Renaissance: `Akka-uct`, `fj-kmeans`, `reactors`, and `neo4j-analytics` perform by far the most object accesses. `Akka-uct` and `fj-kmeans` show the highest object access rate along with `philosophers`, `neo4j-analytics`, `scrabble`, and `naive-bayes` that follow. All applications are dominated by read accesses, with `fj-kmeans` having 120 reads per one write. On the other hand, `db-shootout` and `neo4j-analytics` have the most balanced R/W ratio. Many Renaissance applications show a considerable degree of shared accesses with `reactors` having the most shared reads and writes. Even though actor frameworks aim to guarantee workload concurrency, their asynchronous non-blocking message passing infrastructure inevitably leads to accessing objects "owned" (allocated) by other threads. Therefore, the degree of shared access for `reactors` and `akka-uct` is justified. Nevertheless, it should be noted that both `reactors` and `akka-uct` are artificial stress-test benchmarks. As such, their observed behavior might not fully represent such frameworks in general. On the contrary, `als`, `log-regression`, `naive-bayes` show negligible shared object accesses, thus potentially denoting data parallelism.

The above analysis makes clear that shared accesses profiling can outline insights regarding the internal data dependencies of the application. Such a property, especially with respect to writes, is of high importance since it is tightly related to the scalability of an application (i.e., on a NUMA system).

4.2.4 Object Metrics Summary.

Herefore, we have surveyed all managed applications with respect to several high-level memory metrics related to object allocations and object accesses. Figure 2 groups those applications by the allocation and access rates, and filters out those below the geometric mean using the heuristic of Equation (1):

$$(\ Allocation\ Rate > Geomean\ )\ \textbf{AND}\ (\ Access\ Rate > Geomean\ ) \quad (1)$$

The left and right circles contain applications that exceed the geometric mean of the object allocation rate and the object access rate, respectively. The intersection of the two circles highlights the applications that are intensive both in terms of object allocations and object accesses. The emerging classification confirms already known trends for Dacapo [12, 15]. However, a small differentiation is observed in terms of absolute numbers as a side effect of the metacircular runtime, the slightly different run configurations, and/or the different (updated) version of the benchmark suite. Unlike previous studies [12, 15], we observe that although the above metrics are necessary, they are not sufficient to properly characterize the memory behavior of a managed application. Thus, the next section enhances our study with microarchitectural analysis of metrics provided by PerfUtil.

4.3 Characterization With Low-Level Metrics

This section analyzes and discusses the findings derived from PerfUtil. The insights provided by such a low-level profile complement the findings of Section 4.2 while deepening the understanding of the Dacapo and Renaissance applications. To perform the profiling with PerfUtil, we followed the same experimental procedure with NUMAProfiler, as described in Section 4.2.

4.3.1 Overview of Hardware Instructions.

Table 2 shows the distribution of Arithmetic (integer and floating point), Branch, and Memory Instructions per benchmark. The collected metrics are presented as a percentage over the total number of retired instructions. Table 2 also reveals the ratio between Memory, Arithmetic, and Branch Instructions as well as whether an application is dominated by read or write accesses. PerfUtil counts the *Total Retired Instructions*, *L1D Reads*, *L1D Writes*, *Branch Instructions*, and thus, the number of *Arithmetic Instructions* is calculated as follows:

$$Arithmetic = Total - L1DReads - L1DWrites - Branch \quad (2)$$

Read and write ratios settle towards read operations, since the L1D read instruction percentage is higher than L1D writes for all applications. However, Renaissance applications show a greater diversity than Dacapo. For instance, `als`, `chi-square`, `movie-lens`, and `naive-bayes` (all belong to the Apache Spark family) are below the percentage of minimum memory instructions observed in Dacapo, while `future-genetic` is beyond the maximum one. Nevertheless, the geometric mean of the percentage of memory instructions (Table 2 - Total Mem.) is, as expected, ~45% and all applications are dominated by read accesses. The L1D R/W ratio tends to be aligned to the R/W Ratio of Table 3 even though minor misalignments are visible; probably due to the "noise" introduced by the VM infrastructure. Note that the observed total instructions of a managed application are essentially a mix of instructions from the application and the VM itself. Since there is no obvious way to safely estimate and exclude the latter, a co-interpretation of the results derived from the NUMAProfiler and PerfUtil is necessary.

Table 4. Cache/Memory Locality and Pressure in Dacapo & Renaissance

	CPI	BPU MPKI	Misses PKI				Miss Rate [%]				Accesses PK Obj. Op.	
			DTLB	L1	L2	LLC	DTLB	L1	L2	LLC	LLC	Mem.
avrora	**1.2**	**4.4**	**1.6**	**24.3**	**12.7**	0.1	0.3%	**4.4%**	**52.4%**	1.0%	153	2
fop	0.8	1.0	0.9	13.2	5.9	**1.2**	0.2%	2.8%	44.3%	19.8%	**233**	**46**
h2	**1.2**	**2.7**	**1.8**	12.4	**7.2**	**2.6**	**0.4%**	2.6%	**57.9%**	**36.3%**	178	**65**
jython	0.5	0.5	0.4	8.8	2.2	**1.0**	0.1%	2.0%	25.5%	**44.2%**	82	**36**
luindex	0.6	2.0	0.4	5.2	1.9	0.1	0.1%	1.2%	35.7%	6.8%	70	5
lusearch	0.7	1.6	**1.2**	**16.3**	**6.2**	**0.9**	0.3%	**3.8%**	38.0%	14.6%	**189**	**28**
lusearch-fix	0.7	1.6	1.2	16.1	6.2	0.9	0.3%	**3.8%**	38.2%	14.8%	**188**	**28**
pmd	0.8	1.8	**2.0**	**17.2**	**7.4**	0.8	**0.4%**	**3.7%**	**42.9%**	11.2%	**229**	26
sunflow	0.6	**2.5**	0.5	7.6	2.3	0.7	0.1%	1.8%	30.5%	**31.3%**	48	15
xalan	**0.9**	1.6	1.0	**19.6**	6.2	**0.9**	0.2%	**4.0%**	31.5%	14.9%	**181**	27
GEOMEAN - D	**0.8**	**1.7**	**1.0**	**12.8**	**5.0**	**0.7**	**0.2%**	**2.8%**	**38.6%**	**13.8%**	**138**	**19**
akka-uct	0.8	1.4	**1.1**	**24.5**	**6.8**	**2.5**	**0.3%**	**5.5%**	27.9%	36.6%	122	45
reactors	**1.0**	1.2	0.7	17.6	**7.2**	0.8	0.1%	3.4%	**40.7%**	11.8%	**251**	30
als	0.4	0.3	0.1	2.7	0.9	0.3	0.0%	0.8%	31.8%	29.6%	98	29
chi-square	0.6	0.7	0.1	9.5	2.3	1.2	0.0%	2.3%	24.6%	**51.5%**	95	49
gauss-mix	0.5	0.4	0.1	13.8	4.2	**2.2**	0.0%	3.2%	30.6%	**52.5%**	138	**72**
log-regression	0.6	2.4	0.1	4.9	1.0	0.6	0.0%	1.1%	21.0%	**55.1%**	35	19
movie-lens	0.6	1.2	0.4	8.9	3.2	0.9	0.1%	2.3%	**36.1%**	26.8%	204	**55**
naive-bayes	0.5	0.1	0.1	12.8	3.6	2.0	0.0%	3.4%	28.2%	**56.3%**	197	**111**
db-shootout	0.5	0.5	0.2	8.8	3.4	1.5	0.0%	2.1%	**38.2%**	45.5%	**293**	**133**
fj-kmeans	0.8	1.2	0.6	9.0	4.6	**2.7**	0.1%	1.9%	**51.1%**	**58.9%**	5	3
future-genetic	0.8	0.9	0.3	10.0	3.8	0.7	0.1%	1.8%	38.2%	18.3%	105	19
mnemonics	0.6	**2.9**	0.6	14.6	2.6	1.2	0.1%	3.1%	17.7%	46.6%	91	43
par-mnemonics	0.7	**3.1**	0.2	15.2	3.1	1.2	0.1%	3.3%	20.1%	40.7%	105	43
scrabble	**1.5**	**4.1**	**1.0**	**25.3**	4.8	1.0	0.2%	**5.3%**	18.9%	21.8%	162	35
neo4j-analytics	0.7	1.0	0.5	19.7	3.9	1.8	0.1%	4.4%	19.5%	47.0%	65	31
rx-scrabble	0.9	**2.7**	0.3	**20.1**	3.4	1.1	0.0%	3.5%	17.2%	33.1%	117	39
dotty	**1.1**	**4.1**	**2.2**	**29.2**	**9.3**	1.7	**0.5%**	**6.8%**	32.0%	17.6%	**311**	**55**
scala-doku	0.6	1.6	0.6	15.3	**8.7**	1.4	0.1%	3.6%	**56.5%**	16.1%	**295**	47
scala-kmeans	0.6	2.1	0.2	3.8	0.9	0.5	0.0%	0.8%	24.8%	47.9%	26	13
philosophers	0.9	1.7	0.6	15.4	3.6	0.4	0.1%	3.1%	23.5%	10.3%	92	10
scala-stm-bench7	**1.1**	1.6	**1.1**	**21.0**	**7.0**	**2.1**	0.2%	**4.5%**	33.3%	30.0%	169	51
GEOMEAN - R	**0.7**	**1.2**	**0.3**	**12.3**	**3.5**	**1.1**	**0.1%**	**2.7%**	**28.4%**	**31.9%**	**106**	**34**
GEOMEAN	**0.7**	**1.4**	**0.5**	**12.5**	**3.9**	**1.0**	**0.1%**	**2.8%**	**31.4%**	**24.4%**	**116**	**28**

4.3.2 Data Locality & Cache/Memory Pressure. Although the cache hierarchy aims to fill the latency gap between the CPU and main memory, the latter often remains a source of delays in the execution of a program. Due to complex features of modern hardware (e.g., out-of-order execution, multiple cache levels, shared memory, etc.) such characterization lacks a strict definition (or concrete methodology) and can only rely on a multifaceted profile that comprises numerous metrics. However, CPI co-examination along with memory hierarchy and Branch Prediction Unit (BPU) *pressure* and *locality* metrics can reveal useful insights regarding memory behavior. **Misses Per Kilo Instructions (MPKI)** can be used as a global metric of "pressure" because it factors in the total retired instructions [17, 28]. On the other hand, **miss rate** of each memory level can provide an indication of "data locality". In addition, **LLC and memory accesses per kilo object operations** (allocations + accesses) aim to bridge low with high level metrics and are quite indicative regarding data locality. Moreover, note that large pressure on BPU derives from non-predictive control flow, or data-dependent branches, or both. A non-predictive control flow leads to accessing memory locations in a non-deterministic manner, thereby influencing the regularity of the memory access pattern. Additionally, in the case of data-dependent branches, the accessed memory location cannot be predicted leading to irregular memory access patterns. Therefore, a large value of BPU MPKI can indicate irregularities in the

memory access patterns which in the case of a memory intensive application will penalize performance. The following subsections discuss the above metrics which are listed in Table 4 per application. The five highest values of each metric are highlighted as bold.

The larger the MPKI value is, the "heavier" the load for the corresponding memory hierarchy level is. Consequently, this set of "pressure" metrics can be used to assess the memory-bound degree in correlation to other applications. LLC MPKI reveals that an application's object allocation and access intensiveness are not necessarily reflected in main memory pressure which is counter-intuitive. For instance, in **Dacapo**, `sunflow` is the most intensive application in terms of object allocations and object accesses (recall Section 4.2.1). However, Table 4 shows that the largest pressure on main memory among the Dacapo benchmarks is caused by `h2`. On the contrary, `sunflow` seems to put the least pressure, among the multithreaded applications, on the memory hierarchy as the LLC/Memory pressure and CPI metrics. This is justified by the good spatial and/or temporal locality of `sunflow` working data. `Sunflow's` behavior can also be observed in LLC and memory accesses per kilo object operation ratio which are below the geomean and among the lowest. Consequently, the accesses per kilo object operation metrics are quite indicative regarding the locality of working data by comparing object operations against actual cache/memory pressure. `Fop` is the most LLC and main memory intensive among the single-threaded applications (`fop`, `jython`, `luindex`), which is also confirmed by its CPI. It is notable that although `avrora` and `pmd` are the most LLC intensive applications, they finally put low pressure on memory denoting that their data set successfully fits into the larger LLC (compared to L2). After examining LLC and memory pressure metrics, `avrora's` CPI seems to be affected more by BPU, DTLB, and cache rather than main memory (see Table 4). High DTLB pressure of `lusearch` and `xalan` probably is related to their large objects.

Renaissance: As can be observed in Table 4, `reactors`, `scrabble`, `dotty`, and `scala-stm-bench7` have high CPI values. Such a fact could imply stalls due to memory and consequently memory-boundness. However, this observation contradicts with Figure 2 where only `scrabble` and `scala-stm-bench7` seem to be "object allocation and access intensive". Therefore, the assessment of memory intensity cannot rely only on object-level metrics. An application might be memory-bound due to other reasons (i.e., lack of memory locality), even though it does not significantly allocate or access objects. In particular, `reactors` shows 2.5x more LLC accesses per kilo object operations than the geomean which implies lack of data locality, while `dotty` shows very high BPU MPKI which indicates irregularity in memory access patterns (recall the first paragraph of Section 4.3.2 that explains how the BPU MPKI is related to irregular memory access patterns). Similarly, `par-mnemonics` has been classified as memory-bound [23], however its CPI is way below 1; hence, the memory is not the most decisive factor for performance of this application. In case of `als`, the inspection of the CPI value confirms that it is compute-bound. `Movie-lens` which has been also classified as compute-bound [23], it lacks locality as it shows 2x more LLC accesses per kilo object operations than the geomean. Hence, the performance of this application is rather influenced also by memory.

Moreover, `akka-uct`, `gauss-mix`, `naive-bayes`, `db-shootout`, `scrabble`, `neo4j-analytics`, and `philosophers` are in the intersection of Figure 2; hence it is expected to be memory intensive applications. Nevertheless, they diversify as they do not put pressure on both the LLC and memory. `Akka-uct`, `gauss-mix`, `naive-bayes`, `db-shootout` and `neo4j-analytics` put significant pressure to memory, as expected. On the contrary, `scrabble` and `philosophers` put significant pressure only up to the LLC. It is very likely that `scrabble` and `philosophers` either benefit from locality in LLC or/and have a smaller working data set that fits into LLC. Although, we cannot safely estimate the exact reason for each, note that low LLC and memory accesses per kilo object operation of `philosophers` indicate good data locality. On the other hand, `scrabble` shows high BPU MKPI, a symptom of irregular memory access patterns. Irregular memory access patterns in `scrabble` are also reported by [21], based on the fact that this application deploys a centralized `HashSet` data structure for its working data[4]. In addition, `rx-scrabble` (that implements the same algorithm as `scrabble` but uses an alternative framework), `dotty`, `mnemonics`, `par-mnemonics`, `scala-kmeans`, `philosophers`, and `log-regression` are candidates for irregular memory patterns due to their high BPU MPKI.

A similar diversity is observed for the applications in the right circle of Figure 2. `Fj-kmeans` and `scala-stm-bench7` put significant pressure on both LLC and memory, while `future-genetic` stresses only the LLC. `Fj-kmeans` is the second most intensive application in terms of object accesses, it is a read-dominated application, and according to those data it seems to benefit from data locality since it has the lowest LLC accesses per kilo object operations. Nevertheless, it is notable that [21] characterizes `fj-kmeans` as data locality bound in the context of a NUMA system. Considering this fact, we observe that indeed, this application benefits from data locality in a unified LLC potentially due to limited cached data size. However, this is not true in a distributed LLC environment (like a NUMA system) where cache coherency protocols significantly impact the locality of data in the LLC. The "object allocations intensive" `chi-square` and `scala-doku` do not put

[4]The memory access location is unpredictable in a `HashSet` as it is calculated by a hash function.

significant pressure neither to LLC nor to Memory, since object allocation operations are 3-5 orders of magnitude lower than object accesses. `Log-regression`, `naive-bayes`, `gauss-mix`, and `chi-square`, which are Spark applications, show greater than 50% LLC Miss Rate. Such poor data locality inevitably brings to the spotlight the effect of the Spark engine when co-located with worker threads over the same limited CPU and cache resources. However, only `gauss-mix` and `naive-bayes` end up with high memory pressure among the aforementioned applications. `Akka-uct` has lower CPI than `reactors` although the former has almost three times more accesses to main memory. This counter-intuitive observation is due to the domination of memory instructions in `reactors` (see Table 2), and because it has 2x more LLC accesses per kilo object operations than `akka-uct`. Throughout this comparison, the high complexity of memory behavior analysis is highlighted, thereby denoting that the memory overhead can be derived by any component of the stack.

4.4 The Benefits of Multifaceted Profiling

The above characterization of each application according to its memory behavior is illustratively summarized in Figure 3. The left part of this figure depicts the "view" obtained by each profiling tool individually, while the right part illustrates the "view" that is achieved by co-utilizing those tools. This figure demonstrates the benefits of multifaceted profiling by putting aside and performing a perception-wise comparison between the left with the right part. For example, the "view" of NUMAProfiler indicates that `sunflow` and `philosophers` are memory intensive applications, however as it is turned out by the multifaceted profiling they are neither LLC nor main memory intensive. On the other hand, `dotty`, `fj-kmeans` or `scala-stm-bench7` put considerable pressure on the LLC and the main memory, even though they do not perform much object allocations. Therefore, it is clear that the proposed methodology broadens the profiling view, and avoids misconceptions as well as blind-spots; hence, it provides new opportunities for more effective profiling of managed applications.

4.5 How to Navigate through the Numbers

To exemplify the methodology explained in Section 4.1, we use as an example of a bottom-up analysis the `scrabble` benchmark from the Renaissance suite. By examining the results from the low-level profiler in Table 4, the first metric we focus on is that of the CPI. In general, the larger the CPI the slower the benchmark is. In the case of `scrabble`, we see that the CPI is high compared to the other benchmarks (1.46) which means that this benchmark for some reason(s) does not execute fast. The next step is to discover why `scrabble` behaves this way. For this, we typically check the three main factors that affect performance on modern processors: branch misprediction ratio, cache miss ratio, and TLB miss ratio. As shown in Table 4, scrabble has high BPU (4.11), DTLB (1.05), L1 (25.29), and L2 (4.79) MPKI which justify the high CPI. At this stage, we conclude that `scrabble` puts pressure both on the CPU's front-end (branch predictor) and on the back-end (memory subsystem) which means that the benchmark has significant and unpredictable control flow divergence which may influence also its behavior when accessing memory. Although the MPKI is high for L1 and L2, we observe that the LLC although it has a miss high rate, it is not amongst the worst performers which means the following: 1) either `scrabble` is not too memory intensive and its dataset can fit into the caches, or 2) it is intensive but the hardware prefetcher does a good job in fetching the correct data upon LLC misses. In order to understand in which category `scrabble` belongs, it is now time to investigate the numbers we obtained from the high-level profiler shown in Table 3. As shown in Table 3, `scrabble` has a fairly average sized dataset (2.9 GB) compared to the rest of the benchmarks and does not have any write shared accesses. If we combine all the findings we had so far by investigating the results produced by the two profilers we understand that `scrabble`: 1) has irregular branch behavior, 2) has irregular memory behavior through the cache hierarchy, and 3) although it is memory intensive it does not put significant pressure beyond the LLC. Therefore, by combining the two profilers we derive that `scrabble` exhibits an irregular memory pattern that does not extend beyond its LLC and has negative effect in its performance. The irregular memory access behavior is probably triggered by unpredictable code paths within the code that access different parts of the caches.

As an example of applying the top-down approach, we use the `sunflow` benchmark from the Dacapo suite, for which, we have prior knowledge from existing works. The findings in Table 3 verify prior work. Indeed, `sunflow` has a high object and data allocation rate; 43K objects/sec and 1.8 GB/sec, respectively. By looking at these numbers we may assume that `sunflow` puts significant pressure into memory which may result in high cache miss ratios and hence low performance. To validate this hypothesis, we contrast the high-level numbers with the low-level ones from Table 4. As we see, although `sunflow` has large object and data allocation rates, its performance is amongst the best in the Dacapo suite since its CPI is very low (0.59). This means that although it is regarded as memory intensive, this does not reflect negatively in its performance since both its L1 and L2 miss ratios are very low. However, we observe that the LLC miss ratio is higher than the rest of the benchmarks which is natural since it has a large data set. Hence, the CPU will fetch the data from memory into the LLC upon request. However, because `sunflow` exhibits good memory locality, as soon as the data enter the cache subsystem (via a miss request or by the hardware prefetcher), it makes good (re-)use of them. Therefore, from the micro-architectural point of view, although `sunflow` fetches data from memory due to

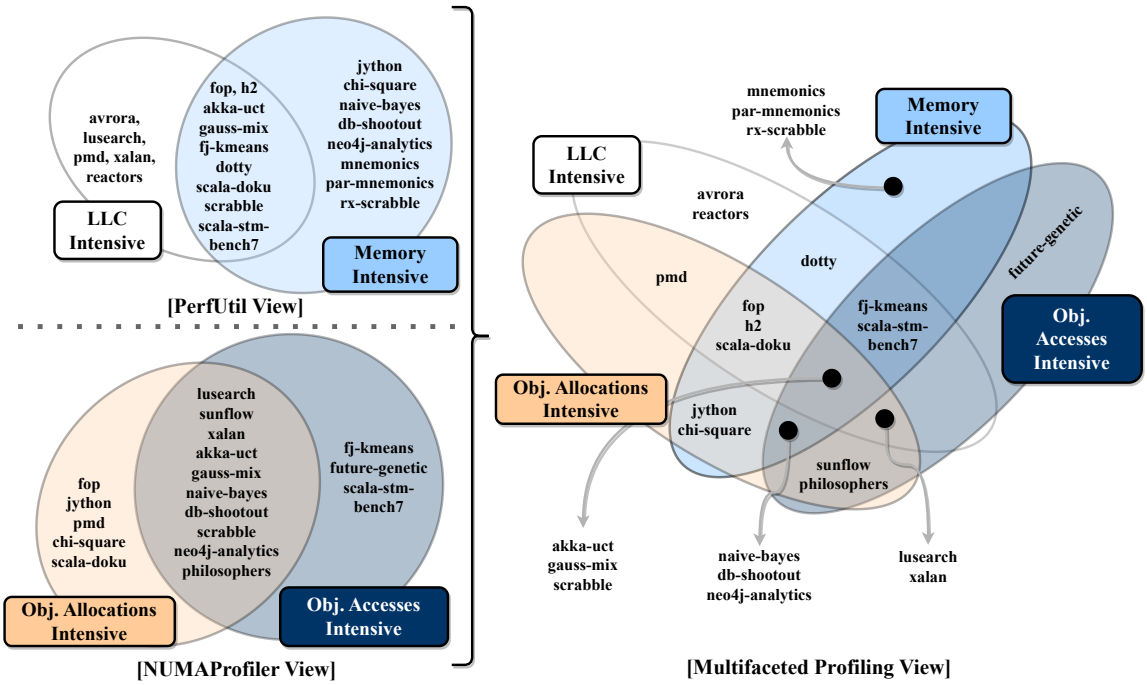

Figure 3. Left: Profiling view from each tool individually. Right: Profiling view by co-utilizing the tools.

Table 5. VM configurations.

VM	MaxineVM v2.9	OpenJDK 8	OpenJDK 11
GC	Semi Space	Serial GC	G1
GC Threads	1	1	8
Generational GC	No	Yes	Yes
Total Heap Size	100GB, Fixed	100GB, Fixed	32GB, Fixed
New Space Size	50GB (From Space)	40GB (Eden Space)	5-60% of Total
Escape Analysis	Not Supported	Disabled	Enabled
Compressed OOPS	Not Supported	Disabled	Enabled

its large dataset, it does it in a way that it does not negatively affect the performance due to extremely good data locality.

4.6 The Impact of MaxineVM on Characterization

This study aims to characterize the memory behavior of the Dacapo and Renaissance benchmark suite using MaxineVM and its profiling tools. Naturally, as Blackburn et al. pointed in their 2006 study [3], "we can draw dramatically divergent conclusions by simply selecting a particular iteration, virtual machine, heap size, architecture, or benchmark" [3]. To quantify the effect of MaxineVM on our study we profile all applications with "perf-stat" on OpenJDK 8 and OpenJDK 11 (Table 5 shows the VM configurations in detail - for machine setup see Table 1). Then, we apply the same characterization heuristics on the obtained results and finally compare and discuss the outcome. The objective of this analysis is to understand if the benchmarks exhibit the same behavior across different MREs rather than comparing the exact metrics.

For this comparison, we exercised two OpenJDK configurations: 1) OpenJDK 8 with Serial collector is used as the closest configuration to MaxineVM while, 2) OpenJDK 11 with G1 GC is used as the latest compatible version with Renaissance v0.11. In order to bring the OpenJDK configurations closer to the MaxineVM one, we disabled both escape analysis and compressed pointers. In addition, we exercised large heaps to minimize GC interference. Lacking an equivalent to PerfUtil profiling tool in OpenJDK, we perform all profiling measurements for all VMs (including MaxineVM) with `perf-stat`. However, `perf-stat` does not allow fine-grain profiling in order to exclude warmup iterations. To minimize this effect we run each application with increased number of iterations (as in Table 2).

Table 6 presents the comparison of the memory behavior characterization between MaxineVM (baseline), OpenJDK 8 and OpenJDK 11. In addition, Table 7 in Appendix A provides an extended collection of metrics for OpenJDK 11. The memory behavior characterization is conducted by examining the L2 MPKI and LLC MPKI as measured with `perf-stat` in order to classify each application as LLC intensive and/or Memory intensive, accordingly. Each application with a value greater than the geomean of its run configuration is considered as LLC/Memory intensive. In essence, we classify each benchmark by comparing itself with the geomean of all benchmarks in its run. Then, we examine whether the result of this comparison is observed across all configuration runs. For example, if we assess the L2 MPKI metric for `avrora`, we can see that in MaxineVM the value is 12.09 (with a 4.07 geomean), in OpenJDK 8 the value is 12.03 (with a 8.15 geomean), and in OpenJDK 11 the value is 10.52 (with

Table 6. Comparison of the memory behavior characterization with OpenJDK 8 and 11 against MaxineVM.

	L2 MPKI				LLC MPKI			
Benchmark	**MaxineVM**	**JDK 8**	**JDK 11**	**Match**	**MaxineVM**	**JDK 8**	**JDK 11**	**Match**
avrora	12.09	12.03	10.48	8, 11	0.10	0.25	0.16	8, 11
fop	5.42	10.18	10.49	8, 11	1.19	2.01	1.85	11
h2	7.16	7.84	7.41	8, 11	2.49	2.66	2.36	8, 11
jython	2.28	7.08	3.91	8, 11	0.92	3.63	1.18	8, 11
luindex	1.98	3.83	4.55	8, 11	0.20	0.35	0.88	8, 11
lusearch	6.16	4.70	4.59		0.81	2.26	1.31	8, 11
lusearch-fix	6.03	4.96	4.55		0.82	2.30	1.32	8, 11
pmd	7.68	9.62	9.15	8, 11	0.78	2.18	1.39	8, 11
sunflow	2.38	2.40	0.98	8, 11	0.68	0.93	0.18	8, 11
xalan	6.26	10.39	9.41	8, 11	0.87	3.21	2.00	8, 11
akka-uct	6.93	7.01	7.23	11	2.28	2.64	2.61	8, 11
reactors	7.30	16.34	12.51	8, 11	0.78	3.03	1.31	11
als	1.06	1.62	1.28	8, 11	0.29	0.64	0.40	8, 11
chi-square	2.50	5.30	3.09	8, 11	1.23	2.54	1.21	8, 11
gauss-mix	4.24	10.79	2.96	8	2.14	6.18	0.96	8
log-regression	1.39	5.59	5.92	8, 11	0.71	3.05	2.73	
movie-lens	3.38	5.95	4.51	8, 11	0.81	2.00	1.18	8, 11
naive-bayes	4.88	9.51	10.82	8, 11	2.67	5.90	6.33	8, 11
db-shootout	3.26	12.41	7.56		1.30	5.61	2.80	8, 11
fj-kmeans	4.44	16.40	16.80	8, 11	2.23	9.69	9.64	8, 11
future-genetic	3.94	14.13	16.05	8, 11	0.65	2.96	1.80	
mnemonics	2.53	13.24	7.07		1.12	6.46	2.80	8, 11
par-mnemonics	3.06	12.93	7.65	8	1.14	6.20	2.73	8, 11
scrabble	5.49	13.30	8.57	11	1.04	5.37	2.22	8, 11
rx-scrabble	3.31	10.48	7.84		1.12	4.67	2.70	8, 11
scala-doku	8.44	5.95	9.43	11	1.30	0.79	0.65	
scala-kmeans	1.10	3.04	4.64	8, 11	0.56	1.50	1.88	8
neo4j-analytics	4.02	10.20	3.55	8	1.77	5.29	1.44	8, 11
dotty	8.87	11.85	8.87	8, 11	1.70	3.23	1.45	8, 11
philosophers	3.59	13.92	8.04		0.33	2.77	0.66	11
scala-stm-bench7	7.12	17.14	15.16	8, 11	1.93	5.61	4.25	8, 11
GEOMEAN	**4.07**	**8.15**	**6.40**		**0.94**	**2.61**	**1.50**	
Match %				**71%, 71%**				**81%, 81%**

a 7.37 geomean). For all configurations, the observed values are significantly larger than the geomean values of their configurations hence they exhibit the same behavior across configuration runs. The "Match" column highlights whether the characterization of a benchmark with OpenJDK 8 and/or 11 *matches* the characterization of the same benchmark with MaxineVM.

The characterization regarding LLC intensiveness matches in 71% (of all benchmarks) between MaxineVM and both OpenJDK 8 and OpenJDK 11. Moreover, the characterization regarding memory intensiveness matches in 81% (of all benchmarks) between MaxineVM and both OpenJDK 8 and OpenJDK 11. These percentage differences inevitably reflect the impact of VM implementations, GC algorithms, etc. on the characterization study. Nevertheless, the OpenJDK 8 and OpenJDK 11 results do not highlight "dramatically divergent" trends and consequently conclusions.

Are the results obtained from MaxineVM transferable to other VMs?: Based on our experiments, the majority of the benchmarks (71%-81%) exhibit the same behavior across different VMs and configurations; albeit with different absolute numbers. For the remaining benchmarks that do not demonstrate the same trends across VMs, we observe that even between runs within the same VM, they behave differently. For example, the results of `scala-doku` and `philosophers` would lead to different characterization even between OpenJDK 8 and OpenJDK 11. In fact, the results in MaxineVM fall in-between OpenJDK 8 and OpenJDK 11. Therefore, it is recommended that for these specific benchmarks that exhibit great sensitivity across different VMs and configurations,

ad-hoc characterization is required in order to draw safe conclusions for a specific study.

5 Related Work

Many research efforts [5, 6, 8, 12, 14–16, 24–26] have aimed to analyze the performance-critical properties of managed applications. Some of them have proposed new profiling tools for the JVM, such as AntTracks [14], AkkaProf [25], FJProf [26], and OXJPerf [16] as well as novel profiling techniques, such as bytecode instrumentation [12], runtime-driven JVM instrumentation [12], application code-wrapping [6], and BottleGraphs [8]. More specifically, Kalibera et al. [12] exploited bytecode instrumentation and runtime-driven JVM instrumentation to study a wide set of concurrency metrics for Dacapo benchmark suite. DuBois et al. [8] leveraged BottleGraphs and studied the exhibited parallelism of Dacapo benchmarks. Lengauer et al. [15] utilized AntTracks to study the memory behavior of Dacapo, Dacapo Scala and SPECjvm2008 benchmark suites. AkkaProf and FJProf are two special-purpose profilers [25, 26] utilized for providing effective profiling metrics for Akka and Fork/Join-based Java applications. Rossa et al. [24] presented P3, a tool for the JVM that exploits bytecode instrumentation and offers a high-level profile related to concurrency, synchronization, etc. Those studies mainly focus on high-level application profiling and metrics, and as a result, they lack correlation with the low-level hardware metrics proposed in our study.

In addition, the choices in the available tooling infrastructure regarding the utilization of Hardware Performance Counters for Java applications are notably limited. The few and rare implementations either lack the ability to perform fine-grain profiling, such as the Oracle Solaris Studio [19], Intel VTune [7, 11], JMH [27] or even are not actively maintained, such as the JRockit [18], and the JikesRVM [2] which both lack support beyond Java 6. One exception is the OJXPerf by Li et al. [16] which is a low-overhead profiler based on perf that binds microarchitectural events to Java objects and targets memory bloats. Nevertheless, the correlation of low with high level metrics that OJXPerf offers is limited to object and method scopes, hence it lacks visibility to the overall memory behavior of the application.

Unlike the aforementioned studies and tools which have not presented a multifaceted approach, Deshmukh et al. [5] deployed perf and Lttng [6] in order to obtain a collection of metrics from the microarchitectural as well as from the runtime layer. However, they focused on the Common Language Runtime (CLR) and on .NET applications.

6 Conclusion

This paper studied the memory behavior of 30 Dacapo and Renaissance applications. For this purpose, a characterization methodology based on a multifaceted profile of a Java application was proposed. The profile is composed of high and low-level metrics collected by two profilers of MaxineVM.

The findings of this work were leveraged to classify the memory behavior of the studied applications into several categories. The analysis complements other related studies by revealing additional insights for the already extensively studied Dacapo applications. Moreover, the study contributes to the memory behavior understanding of the recently introduced Renaissance benchmarks.

This work demonstrates how a characterization methodology that moves away from a single-faceted profiling approach can effectively broaden the analysis perspective by avoiding some misconceptions and blind spots. Both the proposed characterization methodology and the tooling support for the multifaceted profiling can be considered as transferable items that can be applied to other MREs, besides MaxineVM.

Finally, the work presented in this paper aims to initiate new research opportunities, such as profiling studies and optimization approaches, for MREs and to provide the research community with useful insights regarding the memory behavior and characteristics of the most common Java benchmarks.

Acknowledgments

This work is partially funded by the European Union's Horizon 2020 programme under grant agreement No 957286 (ELEGANT) and the European Union's Horizon Europe programme under grant agreement No 101092850 (AERO). Additionally, it is funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee for grant numbers 10048318 (AERO), 10048316 (INCODE), 10039809 (ENCRYPT) and 10039107 (TANGO).

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (Lausanne, Switzerland) (ASPLOS '20). https://doi.org/10.1145/3373376.3378468

[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. 2000. The Jalapeño Virtual Machine. IBM Systems Journal 39, 1 (2000). https://doi.org/10.1147/sj.391.0211

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages,

and Applications (Portland, Oregon, USA) (OOPSLA '06). https://doi.org/10.1145/1167473.1167488

[4] Rui Chen. 2018. Dacapo 9.12 MR1 Release Notes. Github. Retrieved December 18, 2021 from https://github.com/dacapobench/dacapobench/blob/468b86874a2f62c66d111fc871674f935619ca0b/benchmarks/RELEASE_NOTES.txt

[5] Aniket Deshmukh, Ruihao Li, Rathijit Sen, Robert R. Henry, Monica Beckwith, and Gagan Gupta. 2021. Performance Characterization of .NET Benchmarks. In 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). https://doi.org/10.1109/ISPASS51385.2021.00028

[6] Mathieu Desnoyers and Michel Dagenais. 2006. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux. OLS (Ottawa Linux Symposium) (January 2006).

[7] Jack Donnell. 2022. Java* Performance Profiling using the VTune™ Performance Analyzer. Intel Corporation. Retrieved 2022-06-22 from https://www.intel.com/content/dam/develop/external/us/en/documents/219355-vtune-performance-profiling-178112.pdf

[8] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-Threaded Applications. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (Indianapolis, Indiana, USA) (OOPSLA '13). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2509136.2509529

[9] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. 2015. Perf Wiki. Perf Wiki. Retrieved May 4, 2022 from https://perf.wiki.kernel.org/index.php/Main_Page

[10] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (Philadelphia, PA) (USENIX ATC '14).

[11] Intel. 2014. Intel® VTune™ Profiler. Oracle. Retrieved 2022-06-14 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[12] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. 2012. A Black-Box Approach to Understanding Concurrency in DaCapo. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (Tucson, Arizona, USA) (OOPSLA '12). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2384616.2384641

[13] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. 2017. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (Xi'an, China) (VEE '17). https://doi.org/10.1145/3050748.3050764

[14] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (Austin, Texas, USA) (ICPE '15). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2668930.2688037

[15] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (L'Aquila, Italy) (ICPE '17). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3030207.3030211

[16] Bolun Li, Hao Xu, Qidong Zhao, Pengfei Su, Milind Chabbi, Shuyin Jiao, and Xu Liu. 2022. OJXPerf: Featherlight Object Replica Detection for Java Programs. In Proceedings of the 44th International Conference on Software Engineering (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3510003.3510083

[17] Zoltan Majo and Thomas R. Gross. 2011. Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. In Proceedings of the International Symposium on Memory Management (San Jose, California, USA) (ISMM '11). https://doi.org/10.1145/1993478.1993481

[18] Oracle. 2007. About the Oracle JRockit JDK. Corporation, Oracle. Retrieved 2022-06-30 from https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/aboutjrockit.html

[19] Oracle. 2011. Oracle Solaris Studio Software. Corporation, Oracle. Retrieved 2022-06-28 from https://docs.oracle.com/cd/E37069_01/html/E37073/gkmgs.html

[20] Orion Papadakis. 2022. Performance analysis and optimizations of managed applications on Non-Uniform Memory architectures. PhD thesis, The University of Manchester.

[21] Orion Papadakis, Andreas Andronikakis, Nikos Foutris, Michail Papadimitriou, Athanasios Stratikopoulos, Foivos S. Zakkak, Polychronis Xekalakis, and Christos Kotselidis. 2023. Scaling Up Performance of Managed Applications on NUMA Systems. In Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (Orlando, FL, USA) (ISMM '23). https://doi.org/10.1145/3591195.3595270

[22] Orion Papadakis, Foivos S. Zakkak, Nikos Foutris, and Christos Kotselidis. 2020. You Can't Hide You Can't Run: A Performance Assessment of Managed Applications on a NUMA Machine. In Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (Virtual, UK) (MPLR 2020). https://doi.org/10.1145/3426182.3426189

[23] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (Phoenix, AZ, USA) (PLDI 2019). https://doi.org/10.1145/3314221.3314637

[24] Andrea Rosà and Walter Binder. 2020. P3: A Profiler Suite for Parallel Applications on the Java Virtual Machine. In Programming Languages and Systems: 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 – December 2, 2020, Proceedings (Fukuoka, Japan). Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-030-64437-6_19

[25] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. AkkaProf: A Profiler for Akka Actors in Parallel and Distributed Applications. In Programming Languages and Systems, Atsushi Igarashi (Ed.). Springer International Publishing, Cham.

[26] Eduardo Rosales, Andrea Rosà, and Walter Binder. 2020. FJProf: Profiling Fork/Join Applications on the Java Virtual Machine. In Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools (Tsukuba, Japan) (VALUETOOLS '20). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3388831.3388851

[27] Aleksey Shipilev. 2014. Java Benchmarking: as easy as two timestamps. Corporation, Oracle. Retrieved 2022-06-28 from https://shipilev.net/talks/jvmls-July2014-benchmarking.pdf

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems (Pittsburgh, Pennsylvania, USA) (ASPLOS '15). https://doi.org/10.1145/1736020.1736036

Table 7. Extended collection of profiling metrics for Dacapo & Renaissance benchmarks with OpenJDK 11.

									MPKI				Miss Rate			
Benchmark	CPI	Instructions	L1d reads	L1d Writes	Total Mem	Arith.	Branch	BPU MPKI	dTLB	L1	L2	LLC	dTLB	L1	L2	LLC
avrora	1.15	1,145,187,092,622	40.5%	14.6%	55.1%	27.3%	17.6%	5.4	2.7	24.4	10.5	0.2	0.5%	4.4%	42.9%	1.5%
fop	0.94	128,286,649,238	30.7%	11.2%	41.9%	38.7%	19.4%	5.2	1.3	23.8	10.5	1.9	0.3%	5.7%	44.1%	17.6%
h2	1.05	20,236,025,909,823	30.3%	10.7%	41.1%	34.8%	24.2%	3.5	1.5	11.1	7.4	2.4	0.4%	2.7%	67.0%	31.8%
jython	0.56	2,473,114,676,608	31.6%	12.8%	44.4%	35.9%	19.7%	2.0	0.4	12.2	3.9	1.2	0.1%	2.7%	32.2%	30.3%
luindex	0.68	143,227,125,335	29.2%	11.3%	40.4%	40.6%	19.0%	4.2	0.7	11.3	4.5	0.9	0.2%	2.8%	40.4%	19.3%
lusearch	0.79	717,947,226,687	31.3%	13.8%	45.1%	36.5%	18.4%	3.8	0.5	15.9	4.6	1.3	0.1%	3.5%	28.9%	28.4%
lusearch-fix	0.78	698,465,299,117	30.8%	12.6%	43.5%	37.7%	18.8%	3.8	0.4	16.7	4.5	1.3	0.1%	3.8%	27.3%	29.0%
pmd	0.91	432,621,355,210	32.8%	11.9%	44.8%	35.5%	19.7%	4.2	1.2	24.7	9.2	1.4	0.3%	5.5%	37.1%	15.2%
sunflow	0.53	3,687,146,261,831	38.0%	14.2%	52.2%	34.9%	13.0%	2.7	0.2	5.5	1.0	0.2	0.0%	1.1%	17.8%	18.0%
xalan	0.96	2,266,685,709,977	31.7%	13.2%	44.9%	35.2%	19.9%	4.1	0.8	23.5	9.4	2.0	0.2%	5.2%	40.0%	21.2%
akka-uct	0.75	10,264,433,536,567	35.4%	13.9%	49.3%	35.5%	15.2%	1.5	2.0	14.5	7.2	2.6	0.4%	2.9%	49.8%	36.2%
reactors	1.11	2,114,330,887,997	34.3%	11.7%	46.0%	35.2%	18.8%	1.6	1.1	21.5	12.5	1.3	0.2%	4.7%	58.0%	10.4%
als	0.40	4,134,332,529,381	23.4%	7.4%	30.9%	55.1%	14.1%	0.5	0.1	3.3	1.3	0.4	0.0%	1.1%	38.5%	30.8%
chi-square	0.52	1,737,602,525,667	26.2%	7.1%	33.3%	44.3%	22.4%	0.9	0.2	9.2	3.1	1.2	0.0%	2.8%	33.6%	39.0%
gauss-mix	0.49	906,503,864,237	28.3%	8.6%	36.9%	46.9%	16.2%	1.3	0.3	12.9	3.0	1.0	0.1%	3.5%	23.0%	32.4%
log-regression	0.89	597,757,071,971	34.5%	6.0%	40.4%	29.1%	30.5%	8.4	0.5	17.1	5.9	2.7	0.1%	4.2%	34.6%	46.2%
movie-lens	0.61	3,591,824,088,054	26.2%	9.0%	35.2%	47.7%	17.0%	2.2	0.5	10.3	4.5	1.2	0.1%	2.9%	43.9%	26.1%
naive-bayes	0.84	563,772,321,159	32.7%	11.3%	44.0%	37.9%	18.0%	1.7	1.0	19.9	10.8	6.3	0.2%	4.5%	54.5%	58.6%
db-shootout	0.60	2,525,962,230,859	23.4%	12.1%	35.5%	46.2%	18.3%	1.2	0.6	14.8	7.6	2.8	0.2%	4.2%	50.9%	37.0%
fj-kmeans	0.85	3,132,997,222,512	32.5%	5.3%	37.8%	46.2%	16.0%	0.9	0.6	23.5	16.8	9.6	0.1%	6.2%	71.4%	57.4%
future-genetic	1.08	620,715,324,282	39.2%	12.2%	51.4%	30.8%	17.8%	3.7	0.8	27.9	16.1	1.8	0.2%	5.4%	57.6%	11.2%
mnemonics	0.59	833,979,961,083	28.7%	13.0%	41.8%	39.8%	18.4%	1.5	0.6	16.8	7.1	2.8	0.1%	4.0%	42.0%	39.6%
par-mnemonics	0.63	828,256,443,028	28.6%	13.3%	41.9%	39.9%	18.2%	1.7	0.6	17.6	7.7	2.7	0.1%	4.2%	43.4%	35.7%
scrabble	0.79	668,856,625,270	31.6%	11.7%	43.3%	37.9%	18.7%	3.2	0.7	16.8	8.6	2.2	0.2%	3.9%	51.0%	25.9%
rx-scrabble	1.00	226,082,596,560	31.5%	12.1%	43.6%	36.8%	19.6%	3.5	0.9	21.1	7.8	2.7	0.2%	4.8%	37.1%	34.4%
scala-doku	0.58	1,125,345,267,058	38.8%	15.7%	54.5%	28.9%	16.6%	1.9	1.0	16.5	9.4	0.7	0.2%	3.0%	57.1%	6.9%
scala-kmeans	0.61	159,361,934,000	36.5%	4.8%	41.3%	43.8%	14.9%	2.4	0.4	9.8	4.6	1.9	0.1%	2.4%	47.4%	40.6%
neo4j-analytics	0.56	4,864,680,739,287	28.3%	8.1%	36.4%	40.1%	23.5%	1.2	0.3	9.2	3.5	1.4	0.1%	2.5%	38.5%	40.4%
dotty	0.94	1,190,402,351,743	30.0%	9.0%	38.9%	40.1%	20.9%	5.4	1.5	25.2	8.9	1.5	0.4%	6.5%	35.2%	16.4%
philosophers	0.91	1,713,641,244,493	32.1%	14.2%	46.3%	34.3%	19.3%	2.1	0.3	13.0	8.0	0.7	0.1%	2.8%	61.8%	8.2%
scala-stm-bench7	1.49	637,561,312,030	31.3%	12.6%	43.9%	37.6%	18.5%	3.7	2.2	29.8	15.2	4.3	0.5%	6.8%	50.9%	28.1%

A JDK11 Results

This appendix contains the Table 7 that presents an extended collection of profiling metrics for Dacapo & Renaissance benchmarks with OpenJDK 11.

Received 2023-06-29; accepted 2023-07-31