MANCHESTER 1824
The University of Manchester

The University of Manchester Research

You Can’t Hide You Can’t Run

Link to publication record in Manchester Research Explorer

Citation for published version (APA):
Papadakis, O., Zakkak, F., Foutris, N., & Kotselidis, C.-E. (in press). You Can’t Hide You Can’t Run: A Performance Assessment of Managed Applications on a NUMA Machine. Paper presented at 17th International Conference on Managed Programming Languages and Runtimes.

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact openresearch@manchester.ac.uk providing relevant details, so we can investigate your claim.

OPEN ACCESS

Download date:20. Apr. 2025

You Can't Hide You Can't Run: A Performance Assessment of Managed Applications on a NUMA Machine*

Orion Papadakis
The University of Manchester, United Kingdom
orion.papadakis@manchester.ac.uk

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

Nikos Foutris
The University of Manchester, United Kingdom
nikos.foutris@manchester.ac.uk

Christos Kotselidis
The University of Manchester, United Kingdom
christos.kotselidis@manchester.ac.uk

ABSTRACT

The ever-growing demand for more memory capacity from applications has always been a challenging factor in computer architecture. The advent of the Non Unified Memory Access (NUMA) architecture has achieved to work around the physical constraints of a single processor by providing more system memory using pools of processors, each with their own memory elements, but with variable access times. However, the efficient exploitation of such computing systems is a non-trivial task for software engineers. We have observed that the performance of more than half of the applications picked from two distinct benchmark suites is negatively affected when running on a NUMA machine, in the absence of manual tuning. This finding motivated us to develop a new profiling tool, so called PerfUtil, to study, characterize and better understand why those benchmarks have sub-optimal performance on NUMA machines. PerfUtil's effectiveness is based on its ability to track numerous events throughout the system at the managed runtime system level, that, ultimately, assists in demystifying NUMA peculiarities and accurately characterize managed applications profiles.

CCS CONCEPTS

• **Software and its engineering → Software design engineering**.

KEYWORDS

NUMA, JVM, Performance characterization

ACM Reference Format:
Orion Papadakis, Foivos 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 (MPLR '20), November 4–6, 2020, Virtual, UK*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3426182.3426189

*Work-in-Progress paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MPLR '20, November 4–6, 2020, Virtual, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8853-5/20/11…$15.00
https://doi.org/10.1145/3426182.3426189

1 INTRODUCTION

Until recently, multi-core processors have been aggregating processing cores with uniform access latency to the single memory unit of the system. However, as their number increased enormously, the available bandwidth per core has decreased and, thus, the scalability of such designs has come to an abrupt end. To work around the scarce memory capacity and bandwidth limitation, modern multiprocessor designs are based on a non-uniform memory access (NUMA) design. In a NUMA system, cores are clustered into nodes. Each node has a memory controller and is interconnected with other nodes using high speed interconnection links (e.g. Intel's QuickPath (QPI) [16], AMD's HyperTransport [2], etc.). As such, a core can access any memory attached to the multiprocessor, but with non-uniform access latency, since the latency depends on the memory location of the data being accessed.

The efficient exploitation of a NUMA machine is a non-trivial task for software engineers. Typically, a NUMA efficient application needs to limit the amount of remote memory accesses in order to avoid penalizing its performance. For example, an application hosted by a Managed Runtime Environment (MRE), which is unaware of the underlying NUMA topology, is very likely to lead to scattered objects across the different nodes of the NUMA machine and, thus, to excessive amount of remote memory accesses. To motivate towards our objective, Figure 1 briefly presents the performance increase (i.e., blue columns) and decrease (i.e., red columns) of a set of benchmarks, picked from the Dacapo and Renaissance benchmarks suites, by comparing overall performance when utilizing one (i.e., non-NUMA) and two NUMA nodes (i.e., NUMA) of the same machine. A quick glance over the plotted performance values indicates that NUMA mostly results in slowdowns on managed applications. In particular, we notice that **16 out of the 26 benchmarks performed worse on the NUMA machine**. Thus, the observation that more than half (about 60%) of the selected benchmarks when run on a NUMA machine performed worse, motivated us to study, characterize, and better understand NUMA architecture peculiarities.

Recent studies on managed runtimes running on NUMA machines are focused either on characterizing garbage collector's scalability bottlenecks or on introducing various NUMA-aware thread scheduling and memory management policies in the managed runtime system [5, 12–14, 25, 26]. Two other studies [17, 22] have characterized managed applications based on the allocation rate, object layout and garbage collection metrics, as well as they provided a methodology to evaluate concurrency, object synchronization,

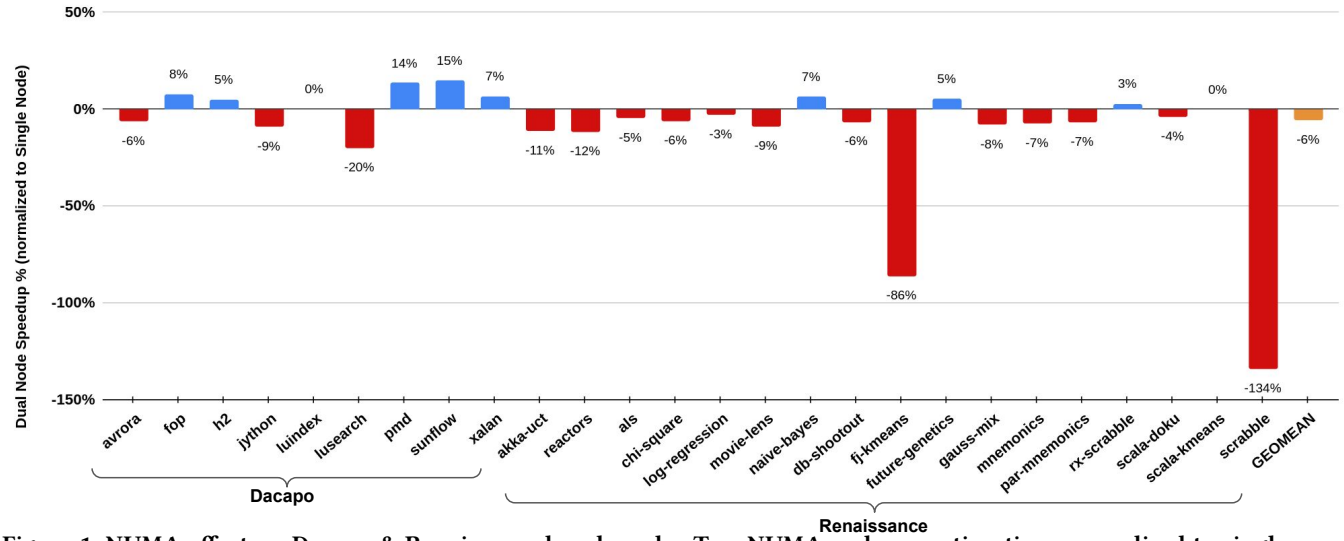

Figure 1: NUMA effect on Dacapo & Renaissance benchmarks. Two NUMA node execution time normalized to single, non-NUMA node. Columns in red color denote slowdown.

and shared memory accesses. Additionally, there are some older studies that utilize HW counters like we do in this work [18, 24, 29]. However, none of those studies provides a characterization on such a wide range of managed applications (from both Dacapo and Renaissance benchmark suites) on top of NUMA machines. Finally, several profiling infrastructures [6, 21, 28, 30] are available for managed runtimes. However, they are either task-specific (ROLP and FJProfiler) or not built for NUMA systems (JProfiler, AntTracks), as well as none of them supports hardware counters utilization.

This work augments MaxineVM [20], a state-of-the-art research managed runtime system, with **PerfUtil, a new profiling tool**. PerfUtil provides access to the hardware-specific performance counters of the system, from the runtime layer, and enables application profiling. In detail, the contributions of this paper are articulated to the following:

- Demonstrate the need for better tooling support to understand the behavior of managed applications running on NUMA systems.
- Implement PerfUtil; a new profiling tool that tracks hardware performance counters at the runtime layer.
- Perform a detailed experimental evaluation on the performance characteristics of a set of benchmarks from both Dacapo and Renaissance benchmark suites.

2 METHODOLOGY

This section presents our methodology for identifying correlations between micro-architectural events and the performance of managed applications, when running on NUMA machines. Towards that objective, we exploit Java benchmarks from Dacapo and Renaissance benchmark suites on a modified version MaxineVM and collect hardware-related metrics from a two-node NUMA machine.

Table 1: NUMA machine setup.

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

2.1 NUMA Machine Setup

Table 1 lists the hardware and software parameters of our experimental infrastructure. We used two configurations: the "Single Node" and "Dual Node" (see Table 2). The former models an 8-core unified memory access system, while the latter enables non-uniform memory accesses (local and remote) between two NUMA nodes. Note that, on the dual node setup we utilize the same amount of cores (8 cores), being evenly distributed across the nodes, and disable the Intel's HyperThreading technology to bound computation capacity that, otherwise, results to performance variation. The default NUMA allocation policy (MPOL_DEFAULT) was used along with page migration for the Dual node configuration.

Performance Monitor Units. Modern multi-processors are equipped with several performance monitor units, which enable tracking of micro-architectural-specific events. In particular, they can be either comprised of a set of fixed (i.e., measure a specific event) or programmable counters. There is a large collection of Performance Monitor Units (PMUs) especially in the context of a modern NUMA system which contains several "individual" modules. Figure 2 abstractly depicts a NUMA node of the Sandy Bridge

Table 2: NUMA machine configurations.

	Single Node	Dual Node
Num of CPUs	1	2
Num of Available Cores	8	16
Num of Utilized Cores	8	8
LLC Size (MB)	20	40
Memory Controllers	4	8
DRAM Size (GB)	192	384
Java Heap Size (GB)	192	384
HyperThreading	off	off

architecture, which is used in this paper. The several sub-modules of each NUMA node are not only shared into the scope of a single node but also system-wide within the multiple nodes of the system. NUMA unlocks scalability by creating a shared pool of individual resources aggregated around a low-latency high-bandwidth two-way interconnect (QPI for Intel) across all NUMA nodes.

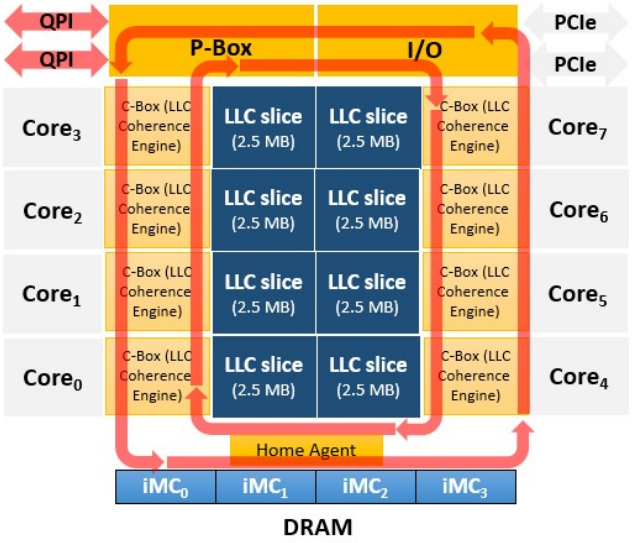

Figure 2: A typical NUMA node architecture.

The Uncore. Apart from the cores, a typical Intel NUMA chip comprises of several modules, the so-called Uncore. Parts of the Uncore are the *B-Box*, the *C-Box*, the *M-Box*, the *P-Box*, the *S-Box* and many more which are out of the scope of the current work [10]. The B-Box and C-Box contain the LLC slices along with the cache coherence infrastructure. The M-Box contains the integrated memory controllers (iMC) as well as the Home Agent which interfaces the iMCs with the interconnect. The P-Box contains the QPI interfaces responsible for the communication with the other NUMA nodes. The S-Box is the physical QPI interconnect module.

Apart from the traditional PMUs, the Uncore is equipped with PMUs with varying amount of counters and, thus, creating more detailed profiling opportunities. Each counter can monitor a wide pool of events, as defined from the silicon manufacturer (e.g. cache access/miss, retired instructions, predictions, etc.).

2.2 Benchmarks Overview

For our analysis we use benchmarks from the traditionally established DaCapo [3] benchmark suite and the state-of-the-art Renaissance [27] benchmark suite. The DaCapo benchmark suite has been heavily used to study and evaluate different garbage collection algorithms, while the Renaissance suite aims to provide more concurrent and modern workloads. It is also worth noting that although the memory footprint of the DaCapo benchmarks is significantly below the memory capacity provided by a typical NUMA system, they have been widely used in many NUMA related studies [1, 13, 14]. An alternative would be to customize e.g. a Big Data framework such as Apache Spark or Flink to achieve a memory footprint of hundreds of GB. However, we consciously preferred to use standardized benchmarks to avoid the uncertainty introduced by customized applications and workloads. Unfortunately, due to some instabilities in MaxineVM, the VM we augment and use for our study, we were not able to use all the benchmarks from the two benchmark suites. As a result we use 9 out of the 14 DaCapo benchmarks and 17 out of 25 Renaissance benchmarks, creating a set of 26 benchmarks in total. A brief description of the key characteristics of the selected benchmarks suites follows.

Table 3: Benchmarks configurations.

	Benchmark	Input Size	Iterations	Allocated Obj. Size
Dacapo	avrora	large (max)	30	410 MB
	fop	default (max)	50	220 MB
	h2	huge (max)	20	24 GB
	jython	large (max)	30	17 GB
	luindex	default (max)	50	30 MB
	lusearch	large (max)	30	12 GB
	pmd	large (max)	30	1.5 GB
	sunflow	large (max)	30	7 GB
	xalan	large (max)	30	12.5 GB
Renaissance	akka-uct	N/A	34	40 GB
	reactors	N/A	20	15 GB
	als	N/A	40	3 GB
	chi-square	N/A	70	3 GB
	gauss-mix	N/A	50	13 GB
	log-regression	N/A	30	1.5 GB
	movie-lens	N/A	30	12 GB
	naive-bayes	N/A	40	12 GB
	db-shootout	N/A	26	35 GB
	fj-kmeans	N/A	40	18 GB
	future-genetic	N/A	60	2 GB
	mnemonics	N/A	26	12 GB
	par-mnemonics	N/A	26	12 GB
	scrabble	N/A	60	3 GB
	rx-scrabble	N/A	90	500 MB
	scala-doku	N/A	30	4 GB
	scala-kmeans	N/A	60	200 MB

Table 3 lists the selected benchmarks, along with the exploited input data size, the number of iterations and the allocated objects size. We used the largest possible data input size provided by each benchmark in Dacapo. Additionally, the number of iterations was selected based on a prior established number able to eliminate warmup's non-determinism [22], which we increased by 10, to achieve a longer end-to-end execution time. Unfortunately, Renaissance do not provide an option of setting the input size. Regarding

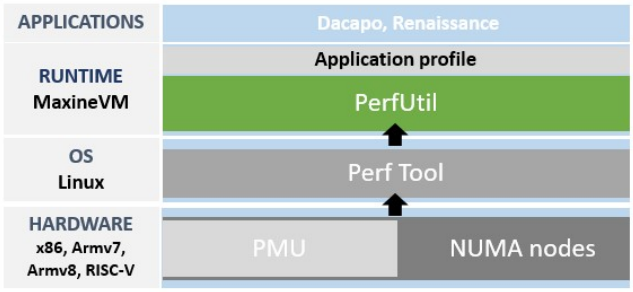

Figure 3: The proposed profiling infrastructure.

the number of iterations in the Renaissance suite we use the default increased by 10.

2.3 PerfUtil Profiler

To achieve high precision in our measurements and remove as much of the runtime system's noise as possible we augment MaxineVM [20], a metacircular research VM written in Java [1,2], with the PerfUtil module. PerfUtil interfaces to the `libperf` OS-library (Figure 3) and enables the monitoring of micro-architectural performance events at the runtime layer. This gives us fine control over when to start or stop the measurements at runtime, e.g., we can start measuring after X full garbage collections, or Y method compilations, etc. To validate our implementation of PerfUtil, we have compared its results against the stand alone version of *perf-stat* tool [31].

PerfUtil is a quite modular and flexible tool as such to be able to model from a single performance event to more complex configurations. PerfUtil also supports perf event groups and monitor scope. The notion of the group ensures that a set of architecturally related events are all simultaneously measured for the same period of time. As such, their values are directly comparable. On the other hand, the monitor scope allows an event to be monitored either per thread, per core or both. Using the thread scope, we bind the monitoring events at VM-level and, thus, reduce the noise introduced by the rest of the processes running simultaneously on the system. Additionally, a thread scope can also be combined with a core scope resulting in monitoring a specific thread in a specific core. However, there are some PMUs that cannot be utilized per thread such as the those of the memory controller. Consequently, a per core scope is necessary.

2.4 Tracked Events

The key feature of NUMA machines is that the memory access latency may vary significantly depending on the location of the memory being accessed. For that purpose, we chose to monitor NUMA-related events that could help us understand the kind of accesses performed by the applications (i.e. local access vs. remote access). In particular, the PerUtil tool was exploited to monitor the following events:

- **Last Level Cache (LLC) miss:** A LLC read or write miss in a non-NUMA architecture leads inevitably in accessing the main memory (i.e. the DRAM). However, this is not the typical procedure for the NUMA systems. On a NUMA machine, a LLC miss occurs when the requested data has not been found in the Local LLC, although, they could potentially be retrieved from a remote node LLC. In particular, on a LLC miss, the caching agent, initially, broadcasts the miss event to the other nodes via the interconnect, before accessing the memory. In case that a remote LLC slice has a valid copy of the requested data, the LLC miss is a *remote LLC hit* and, thus, avoids the main memory access. On the contrary, the local or remote DRAM memories have to be accessed. Therefore, a LLC miss in a NUMA machine corresponds to a *local* LLC miss [15]. Finally, note that the process of fetching the data from the remote node significantly penalizes the performance of the application.
- **Node Read/Write:** This event refers to the number of memory accesses (either reads or writes) served locally [19].
- **Node Read/Write miss:** This event refers to the number of LLC and memory accesses served by a remote NUMA node. Note that, this event should not be confused with remote memory accesses [19].
- **Memory controller:** This event refers to the read and write requests arriving to the memory controller (iMC) in a system-wide scope. In particular, each memory controller monitors the incoming requests issued by the application, the managed runtime, the OS or by any other running process on the system [10].

Although for the purpose of this study we traced a specific set of performance events, the PerfUtil profiling tool can be seamlessly exploited to monitor any event that is provided by the underlying hardware architecture (e.g. branch predictions, L1 or L2 hit/miss, etc.).

2.5 Measurement Methodology

PerfUtil events. As mentioned in Section 2.3 the events collection is dynamically controlled by the extended managed runtime system. PerfUtil has been configured to monitor the events of choice. However, in many cases it was impossible to concurrently monitor all the events due to the limited amount of the physical hardware counters. Thus, to avoid multiplexing and its shortcomings (i.e., precision loss), we perform more than one runs per benchmark; each configured to gather a different set of metrics. All the events, except for the memory controller where this is not possible, are measured using the per thread monitoring scope to isolate the results from other processes' potential interference.

Thread count control. The number of threads was set to eight to match the number of utilized cores in both the Single and Dual node configurations. However, some benchmarks do not comply with such an option (e.g., `avrora` inevitably spawns 32 threads when run with the large input size). For those, we adhered to the default thread amount configuration set by the application. Note also that even the single threaded applications utilize multiple threads due to the five JVM-internal threads, which are mostly idle during application code execution.

Heap size. The JVM heap size was configured to 192 GB for Single Node and to 384 GB for Dual Node to match the underlying physical memory capacity. In practice, this means that no garbage

[1] MaxineVM's modularity and ease to extend was the key enabling factor for this decision.
[2] https://github.com/beehive-lab/Maxine-VM

collections are triggered due to running out of heap space. However, all benchmarks perform an explicit GC before every iteration as part of the benchmark suites' harnesses. In our measurements we include the events triggered by these GC operations as well.

Gathering the measurements. Each measurement session covers the corresponding application's end-to-end execution including all iterations and the GC operations between them. Warm-up iterations are also included, however due to the chosen large amount of iterations they have a minor effect on final results, thus we chose to include them and provide end-to-end results.

3 ANALYSIS

In this section, we present our experimental results, as derived from the comprehensive performance analysis of two benchmark suites (more details in § 2.2), when running on top of a NUMA machine (more details in Tables 1 and 2).

3.1 Locality Evaluation: LLC misses

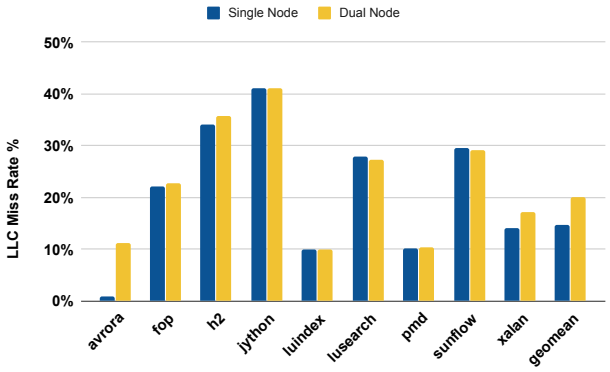

Figure 4: LLC miss rate in Dacapo benchmarks.

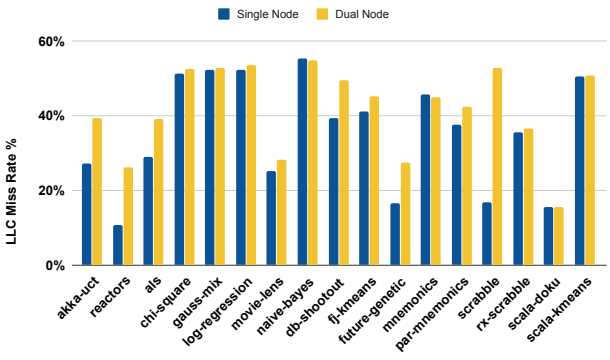

Figure 5: LLC miss rate in Renaissance benchmarks.

This section presents the measured LLC miss rate for the Dacapo and Renaissance benchmarks. On a NUMA machine, an increased amount of LLC misses (or, adversely, remote node hits) constitutes a strong indication of a high volume of shared objects. Additionally, the thread migration policy can also affect the LLC miss rate. For example, consider Thread (T1) is running on core 0 (within Node 0) and is writing Object-A on the Local LLC. However, T1 migrates to another NUMA node (e.g., to Core 3, on Node 1). In that case, if T1 attempts a write operation on Object-A, a LLC write miss will occur (which doesn't happen on the single, non-NUMA setup), the updated data will be written in Core's 3 LLC slice and an update operation to invalidate the copy of Core's 0 will also be triggered (i.e., a cache invalidation).

Figures 4 and 5 present the LLC Miss rate for the Dacapo and Renaissance benchmarks. A key finding is that avrora, xalan, akka-uct, reactors, als, movie-lens, db-shootout, fj-kmeans, future-genetic, par-mnemonics and scrabble applications demonstrate a considerable increase in the LLC Miss rate of the dual node setup, when compared to the single node setup. This behavior is justified for avrora and xalan [17] application due to the high volume of write operations on shared objects. Accordingly, this behavior on the other benchmarks is attributed to the high number of cache invalidations.

The fork/join applications, such as the scrabble (which exhibits the highest increase in LLC miss rate, ranging from 17% to 53%), fj-kmeans, future-genetics and par-mnemonics, have sub-optimal utilization of the available compute resources and thus they cannot take maximum advantage of the available parallelism, which is provided by a NUMA machine [28]. In particular, these applications generate a small number of big, unbalanced tasks with data dependencies and lack of concurrency [28]. That said, our LLC miss rate findings are also a natural explanation of this behavior. Finally, it was counter-intuitive that the akka-uct and reactors benchmarks show high LLC miss rate on the dual node setup since actor frameworks, through the event-driven message-passing mechanism, simplify thread communication and synchronization thus minimizing data dependencies.

Overall, this set of experimental results highlight the importance of having a NUMA capable profiling tool that would assist software engineers to understand and develop an efficient mitigation strategy for handling the excessive amount of share objects that prevents the exploitation of computing capabilities of a NUMA machine.

3.2 Locality Evaluation: Remote Node Accesses

This section analyzes the remote node access rate for Dacapo and Renaissance benchmark suites.

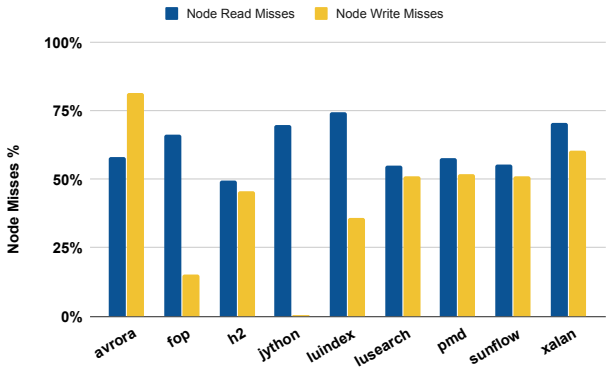

Figure 6: Dacapo node miss rate.

The analysis of remote node access rate in relation to the LLC miss rate provides a better insight about the performance impact of a NUMA machine. As already stated, a remote node access is a

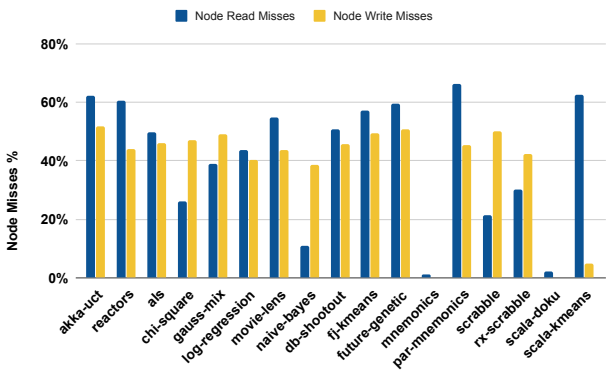

Figure 7: Renaissance node miss rate.

memory operation served by a remote NUMA node, since it has missed on the local memory-hierarchy. We define this behavior as, *node miss* (see § 2.4). Figures 6 and 7 show the *node miss* rate for both *read* and *write* operations, as a percentage of the LLC read and write misses.

A high amount of shared object accesses results in a high node miss rate and in a moderate application speed up when running on NUMA machine. Therefore, even though, remote node access operations are used to increase the application's memory capacity and, thus, avoid accessing the high-latency storage device, they should be carefully handled. For instance, avrora and xalan benchmarks have the highest node write miss rate (81% and 60%, respectively), since they encounter the highest amount of shared write accesses [17]. Additionally, avrora, akka-uct, fj-kmeans and scrabble (and many other benchmarks) show a considerable performance slowdown, when running on the dual node configuration. This behavior highlights the high correlation between performance and node miss rate. Additionally, h2 benchmark has 5% speedup in the NUMA setup (i.e., Dual Node configuration) by showing relatively low remote node accesses (49% and 45% node read and write miss, respectively). As intuitively expected, the single-threaded applications have lower node write miss rate (median value is 2.7%), when compared to multi-threaded application, since they do not exhibit shared written objects. However, it is notable that other benchmarks of relatively high node miss rate (i.e., xalan with the second highest node write miss rate) show considerable speedup in the NUMA setup. Such a fact indicates that, along with NUMA, various layers/factors of a modern stack practically interfere and thus might be decisive for an application's overall performance (see § 4.1).

Overall, the object placement related MRE components (e.g., the Garbage Collector and the threads) influence on local versus remote node memory utilization can possibility affect the application's performance.

3.3 Load Balancing

Many studies highlight the importance of having a load balanced execution of workloads on the available hardware resources [4, 11, 23] of a NUMA system. That said, this section presents and analyzes the load distribution of each memory controller (iMCx) for the employed testbed (eight memory controller, Table 2). In particular, Figures 8 and 9 show the percentage of memory requests served by each controller, over the total amount of memory requests issued by each application. Finally, note that the presented experimental results are an aggregation of the memory requests issued to the controller from both the application-, OS- and runtime-level.

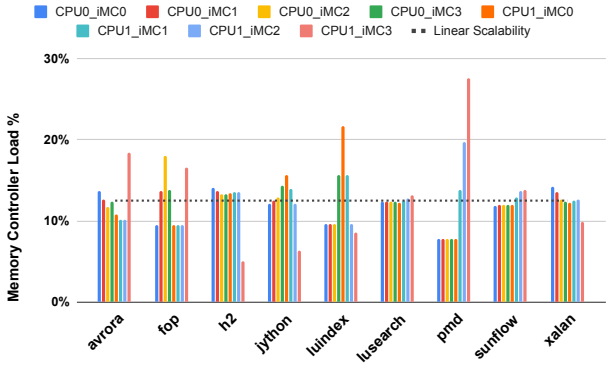

Figure 8: Dacapo memory controller load in dual node.

Optimally, a linearly scalable application should uniformly distribute the memory requests across all memory controllers. In our experimental setup, this is translated to a threshold of 12.5% of the overall memory requests to be served by a single controller (i.e., the dashed line on Figures 8 and 9). Therefore, the closest to that threshold a benchmark is, the more equally distributed the load will be.

Based on our experimental findings, fj-kmeans and scrabble benchmarks have the most unbalanced utilization of the available memory controller. In particular, a single, over-congested memory controller serves 35% and 49% of the overall load for the fj-kmeans and scrabble, respectively. Additionally, this observation verifies our benchmark analysis (§ **??**), in which fork/join applications suffer from sub-optimal forking, which, in turn, leads to uneven task distribution. On the contrary, applications with explicit concurrency and parallelization, such as h2, lusearch, and sunflow, are able to scale very close to the linear scalability threshold.

There are still some benchmarks, in which NUMA performance doesn't seem to be easily explained with the metric used so far. For example, lusearch is totally balanced but slows down in NUMA execution, while pmd has unbalanced utilization of the memory controllers but it achieves to speed up. However, our analysis regarding the OS mechanisms (i.e., page migration and DVFS) that affect the performance of managed applications achieve to adequately explain this abnormal behavior (more details on § 4.1).

4 DISCUSSION

In this section we further discuss OS mechanisms that affect the behavior of managed applications, key findings and, finally, the limitations of our work.

4.1 OS Mechanisms

To highlight the importance of factors external to the application and show how they affect performance, even without always affecting the low-level metrics gathered through PerfUtil, we perform a

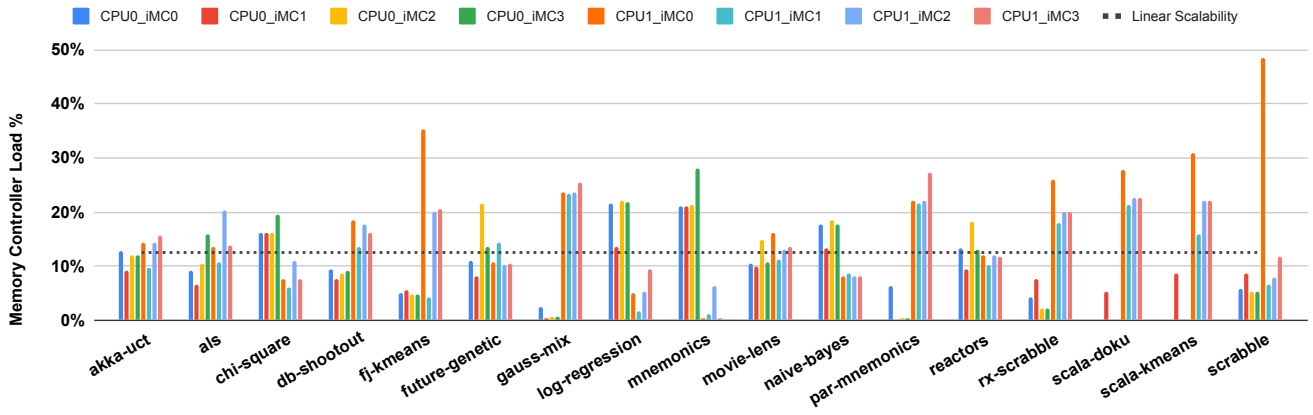

Figure 9: Renaissance memory controller load in dual node.

set of additional experiments by turning off Page Migrations and Dynamic Voltage and Frequency Scaling. We find that both these factors have a considerable effect on the applications execution time.

4.1.1 Page Migration Effect. Page Migration (PM) or AutoNUMA [7] as an OS optimisation aims to enhance performance by periodically moving memory pages across NUMA nodes. The migration algorithm works as follows, the OS periodically invalidates memory pages; thus provoking "artificial" page faults the next time the page is accessed. An artificial page fault handling results in page migration to the NUMA node of the thread responsible for the page fault [7–9]. This way pages get re-shuffled periodically based on the heuristic that the thread that performs the most accesses to a page will end up fetching the page to its NUMA node more times than the rest, thus increasing locality and reducing remote node accesses. However, the overhead of migrating a 4 kB memory page is significantly higher than a remote node access [32]. Therefore, the restrained remote node accesses cost due to a page migration should be *more* than the migration overhead itself, to improve overall performance. Thus, the trade-off *improved locality v.s. the page fault and page move costs* introduced by this mechanism might not always be beneficial.

Table 4 depicts the effect of PM on Dacapo and Renaissance node misses and execution time. The reported percentages are calculated using the following formula for the corresponding value of each column.

$$\frac{Value\ with\ PM\ on - Value\ with\ PM\ off}{Value\ with\ PM\ off} \times 100$$

Negative numbers denote a decrease in the value due to PM and are considered better. We see that in all cases, except for `Xalan`, `Fj-kmeans`, and `Scrabble` page migration manages to reduce the overall node misses. However, it is noteworthy that despite the decrease of the overall node misses PM doesn't have the same effect on the execution time. Only `Fop`, `H2`, `Jython`, and `Scala-doku` seem to benefit from page migration, while most of the other benchmarks perform worse. At a first read these numbers appear to indicate that there is no correlation between node misses and execution time, which is counter-intuitive. This anomaly is explained by the fact that page migration itself comes at a cost. The periodic invalidation of the pages increases the number of page faults which in turn results in additional overheads due to the page fault handling mechanism. Furthermore, whenever a page is moved to a different node an additional overhead for the page copy is payed. The sum of these overheads seems to offset the benefits gained by the reduced node misses in our experiments. Consequently a trade-off is emerging; *in what extent is such a mechanism actually beneficial in the context of MREs and under which conditions?*

Counter-intuitive cases such as *Xalan*, where PM further increases the node misses reflects that overall performance is not only impacted by locality. This makes clear that optimisations targeting only in data locality do not always guarantee improved performance. Other NUMA-related studies agree to that fact as well, while they highlight factors of greater importance such as interconnect contention, memory controller congestion and unbalanced load [4, 11]. Additionally it hints that the OS might not be the proper "place" for NUMA optimisations in the context of managed applications. A NUMA-aware JVM can more effectively avoid hurting locality by taking over its own thread scheduling from the OS [26].

4.1.2 Dynamic Voltage and Frequency Scaling. Dynamic Voltage and Frequency Scaling (DVFS) can significantly improve the performance, but at the same time may skew measurements making correlations between NUMA-related metrics and performance even harder. As shown in Figure 1, sunflow gains 15% speedup on the Dual Node (vs. Single), despite its high node miss rate (about 50%, Figure 6). To better understand that, we monitored operating frequency and revealed that sunflow was running, on average, at a different frequency on the Single Node vs. the Dual Node configuration (i.e. 18% higher). As a result, this finding indicates how ostensibly unrelated optimizations mechanisms may skew measurements and misguide us.

4.2 Take-Away Messages

The main objective of this work is to assist software engineers to better understand the peculiarities of NUMA machines, when

Table 4: Page migration effect on Dacapo and Renaissance.

Benchmark	Node W Misses	Node R Misses	Exec. Time	Benchmark	Node W misses	Node R Misses	Exec. Time
avrora	1%	-1%	0.03%	gauss-mix	-10%	-34%	11.27%
fop	-55%	-46%	-1.08%	log-regression	-36%	-28%	3.86%
h2	-70%	-62%	-9.89%	movie-lens	-17%	-11%	8.93%
jython	-70%	-80%	-4.03%	naive-bayes	-23%	-80%	16.02%
luindex	-76%	-35%	4.43%	db-shootout	-13%	-7%	15.81%
lusearch	-6%	3%	13.12%	fj-kmeans	-2%	24%	48.17%
pmd	-17%	-25%	4.24%	future-genetic	-2%	-1%	1.52%
sunflow	-3%	-2%	1.63%	mnemonics	-100%	-89%	2.60%
xalan	311%	295%	0.03%	par-mnemonics	-39%	20%	3.74%
akka-uct	-9%	-5%	13.37%	scrabble	-8%	88%	36.07%
reactors	-25%	-5%	1.54%	rx-scrabble	-4%	-50%	1.15%
als	-18%	-20%	2.60%	scala-doku	-95%	-70%	-5.15%
chi-square	-9%	-53%	6.12%	scala-kmeans	-48%	23%	0.02%

executing managed applications. To address that objective, this section summarizes our key findings.

- **Threads grouping**: Schedule software threads, which share objects, on the same NUMA node to avoid excessive remote node accesses and cache invalidations. Note that oversubscribing threads to a single NUMA node may have the opposite results, due to increased interconnect connection.
- **Threads spreading**: Spread threads that do not share objects to the available NUMA nodes as much as possible. This will help achieve more uniform utilization of memory controllers.
- **NUMA-aware garbage collection**: Augment GC algorithms with NUMA-aware heuristics to improve objects' data locality and to avoid excessive amounts of remote accesses during the GC phase.
- **High-level programming frameworks**: Augment high-level programming frameworks with NUMA-aware heuristics to avoid data contention. As our experimental results demonstrated, the high variation in LLC miss rate between non-NUMA and the NUMA configurations is a strong indication of a high volume of shared written objects (see § 3.1 and § 3.2).
- **Memory controller utilization**: Avoid high utilization of memory controllers, since it can result to higher performance overhead, when compared to remote node accesses (e.g., `fj-kmeans` 86%, `scrabble` 134% performance overhead, see § 3.3).
- **Thread and data movement management**: Managed runtime should be able to control thread and/or data movements within the available node in order to avoid an unnecessary migration that would penalize performance (see § 4.1).
- **Page migration**: OS level mechanisms appear to do more harm than good in the context of managed applications (see § 4.1.1). Therefore, it is suggested to pass their control over the runtime.

4.3 Limitations

Our test-bed has 192 GB of DRAM per node, while the exploited benchmarks can allocate up to 40 GB. As a result, this work only covers the behavior of applications that are expected to co-exist with other applications in a NUMA machine, since they can't utilize the available resources on their own. A study of larger applications that could utilize the whole machine is expected to unveil even more interesting facts about the nature of NUMA machines and the behavior of managed applications on them. Unfortunately, to the best of our knowledge there is no standardized benchmark suite with benchmarks large enough to utilize a NUMA machine, even with only two NUMA nodes. Therefore, benchmarks with memory footprint from hundreds of GB to tens of TB, ideal for NUMA research, are a necessity.

Currently, PerfUtil can be easily used to correlate low-level metrics with specific phases of the runtime, but unfortunately it's not able to attribute low-level metrics to a specific Java thread, or data structure. As a result, the micro-architectural metrics used to perform this study can only provide an indication about the existence of a bottleneck. Therefore, a high-level study, in the context of managed runtime, able to isolate the sources of inefficiencies is necessary.

5 CONCLUSIONS

Clearly, porting managed applications to a NUMA machine is a non-trivial task. Software engineers have to consider multiple, cross-layer performance events and configurations, in order to be able to understand how the NUMA architecture affects the applications' performance. It is evident though the importance of having a profiling tool that is capable of monitoring a wide range of system events. Towards that objective, this paper presents a new profiling tool, so called PerfUtil. PerfUtil's effectiveness is based on its ability to track numerous events throughout the system that, ultimately, assists in demystifying NUMA peculiarities and accurately characterize managed applications profiles. Finally, we have reported our findings in the context of the traditionally established Dacapo benchmark suite and the state-of-the-art Renaissance benchmarks.

ACKNOWLEDGMENTS

This work was partially supported by EU Horizon 2020 project grants: ACTiCLOUD with ID 732366 and E2Data with ID 780245.

REFERENCES

[1] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-Rationing Garbage Collection for Hybrid Memories. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and*

Implementation (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 62–77. https://doi.org/10.1145/3192366.3192392

[2] AMD. 2019. HyperTransport Consortium. https://www.hypertransport.org

[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)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIXATC'11)*. USENIX Association, USA, 1.

[5] Rodrigo Bruno and Paulo Ferreira. 2018. A study on garbage collection algorithms for big data environments. *Comput. Surveys* 51, 1 (2018). https://doi.org/10.1145/3156818

[6] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. 2019. Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 28, 16 pages. https://doi.org/10.1145/3302424.3303988

[7] Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/

[8] Jonathan Corbet. 2012. Toward better NUMA scheduling. https://lwn.net/Articles/486858/

[9] Jonathan Corbet. 2013. NUMA scheduling progress. https://lwn.net/Articles/568870/

[10] Intel Corporation. 2012. Intel ® Xeon ® Processor E5-2600 Product Family Uncore Performance Monitoring Guide. March (2012), 1–136.

[11] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[12] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2011. Assessing the Scalability of Garbage Collectors on Many Cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems* (Cascais, Portugal) *(PLOS '11)*. Association for Computing Machinery, New York, NY, USA, Article 7, 5 pages. https://doi.org/10.1145/2039239.2039249

[13] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 2013. A Study of the Scalability of Stop-the-world Garbage Collectors on Multicores. *Asplos* (2013), 229–240. https://doi.org/10.1145/2451116.2451142

[14] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC. *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '15* (2015), 661–673. https://doi.org/10.1145/2694344.2694361

[15] JR Goodman and HHJ Hum. 2009. MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects (2009). *Researchspace.Auckland.Ac.Nz* (2009). https://researchspace.auckland.ac.nz/bitstream/handle/2292/11593/MESIF-2004.pdf?sequence=7

[16] Intel. 2020. Intel QuickPath Interconnect: Maximizing multicore processor performance. https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html

[17] 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, 335–354. https://doi.org/10.1145/2384616.2384641

[18] Martin Karlsson, Kevin E Moore, Erik Hagersten, and David A Wood. 2003. Memory system behavior of Java-based middleware. In *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings*. IEEE, 217–228.

[19] Andi Kleen. 2017. perf events (node-loads, node-load-misses, ...). https://www.spinics.net/lists/linux-perf-users/msg03351.html

[20] 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)*. Association for Computing Machinery, New York, NY, USA, 74–82. https://doi.org/10.1145/3050748.3050764

[21] 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, 51–62. https://doi.org/10.1145/2668930.2688037

[22] 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, 3–14. https://doi.org/10.1145/3030207.3030211

[23] 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)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1993478.1993481

[24] Morris Marden. 2001. An Architectural Evaluation of Java TPC-W. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA '01)*. IEEE Computer Society, USA, 229.

[25] Duarte Patrício, Rodrigo Bruno, José Simão, Paulo Ferreira, and Luís Veiga. 2017. Locality-aware GC optimisations for big data workloads. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10574 LNCS. 50–67. https://doi.org/10.1007/978-3-319-69459-7_4

[26] Maria Patrou, Kenneth B. Kent, Gerhard W. Dueck, Charlie Gracie, and Aleksandar Micic. 2018. NUMA awareness: Improving thread and memory management. *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018* (2018), 119–123. https://doi.org/10.1109/SEAA.2018.00028

[27] 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)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[28] 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, 128–135. https://doi.org/10.1145/3388831.3388851

[29] Luiz De Rose, Luiz De Rose, Ying Zhang, Ying Zhang, Daniel A. Reed, and Daniel A. Reed. 1998. SvPablo: A Multi-Language Performance Analysis System. In *In 10th International Conference on Performance Tools*. 352–355.

[30] EJ Technologies. [n.d.]. Java Profiler - JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html

[31] Perf Wiki. 2006-2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org

[32] Rui Yang, Joseph Antony, and Alistair Rendell. 2009. Effective use of dynamic page migration on NUMA platforms: The Gaussian chemistry code on the sunfire X4600M2 system. *I-SPAN 2009 - The 10th International Symposium on Pervasive Systems, Algorithms, and Networks* (2009), 63–68. https://doi.org/10.1109/I-SPAN.2009.127