Scaling Up Performance of Managed Applications on NUMA Systems

Orion Papadakis†, Andreas Andronikakis†, Nikos Foutris†, Michail Papadimitriou†, Athanasios Stratikopoulos†, Foivos Zakkak±, Polychronis Xekalakis┬, Christos Kotselidis†
†The University of Manchester, ±Red Hat, ┬NVIDIA
first.last@manchester.ac.uk,fzakkak@redhat.com,pxekalakis@nvidia.com

Abstract

Scaling up the performance of managed applications on Non-Uniform Memory Access (NUMA) architectures has been a challenging task, as it requires a good understanding of the underlying architecture and managed runtime environments (MRE). Prior work has studied this problem from the scope of specific components of the managed runtimes, such as the Garbage Collectors, as a means to increase the NUMA awareness in MREs.

In this paper, we follow a different approach that complements prior work by studying the behavior of managed applications on NUMA architectures during mutation time. At first, we perform a characterization study that classifies several Dacapo and Renaissance applications as per their scalability-critical properties. Based on this study, we propose a novel lightweight mechanism in MREs for optimizing the scalability of managed applications on NUMA systems, in an application-agnostic way. Our experimental results show that the proposed mechanism can result in relative performance ranging from 0.66x up to 3.29x, with a geometric mean of 1.11x, against a NUMA-agnostic execution.

CCS Concepts: **• Computer systems organization → Multicore architectures**; **• Software and its engineering → Object oriented languages**; **Runtime environments**; **Software design engineering**.

Keywords: NUMA, Scalability, Optimization, JVM, Managed Runtimes, Dacapo, Renaissance, MaxineVM

1 Introduction

The advent of Non-Uniform Memory Access (NUMA) architectures has posed significant challenges with regard to the performance scalability of managed applications. The primary reason is that those applications are running on top of a managed runtime environment (MRE) which is typically oblivious of the NUMA characteristics. Therefore, two research questions that arise are: *Which are the limiting factors of MREs in the context of NUMA systems? Which parts of MREs should be enriched with NUMA awareness?*

In recent years, several studies have attempted to address the aforementioned questions in the context of the Java Virtual Machine (JVM) [2, 9, 10, 22]. These research studies have mainly focused on specific characteristics of the Garbage Collector (GC) towards enhancing its scalability. However, the performance of Java applications on NUMA architectures is not only a subject of GC scalability. To comprehend this, we have compared the NUMA performance (excluding GC time) of the Dacapo [3] and Renaissance [24] benchmark suites against their non-NUMA execution, when running on top of MaxineVM [15]. As shown in Figure 1, the *mutation* execution time can be significantly penalized up to 133%.

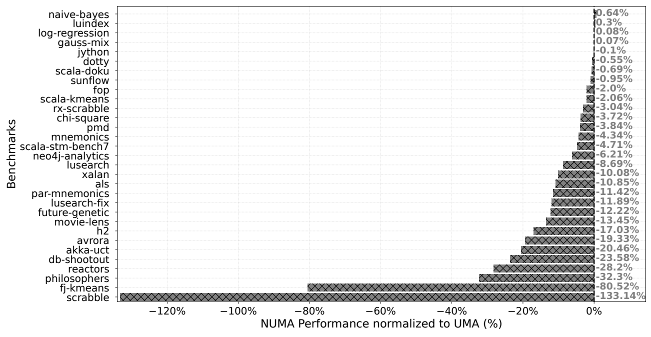

Figure 1. NUMA effect on Dacapo & Renaissance.

In addition, other studies have shown that a number of Dacapo benchmarks (e.g., `fop`, `jython`, `luindex`) lacks properties (i.e., parallelism/concurrency) that are prerequisites for scalability [7, 14]. Thus, those applications may not benefit from NUMA architectures, and they may be significantly penalized even in the optimal case of having a linearly scalable GC. As a result, it is of utmost importance to have an in-depth understanding of the circumstances under which the performance of managed applications can scale up on NUMA systems.

In this paper, we follow a complementary approach that studies the behavior of Java applications on NUMA architectures during mutation time. In particular, we perform a study that characterizes the NUMA scalability of several

ISMM '23, June 18, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595270

Dacapo and Renaissance applications. Our study is undertaken in MaxineVM [15]; an open-source research VM for Java that encapsulates two tools: the i) NUMAProfiler [20] for application-layer profiling, and ii) PerfUtil [20, 21] for microarchitectural profiling. In our study, we employ those tools to obtain an effective profile for the application properties that relate to scalability (e.g., shared object accesses, workload parallelism, data dependencies, data locality, etc.) in the context of a NUMA system. The profiling of the properties enables the classification of a Java application into discrete categories. The resulting classification augments our understanding of the application characteristics, and enables us to draw a conclusion regarding the circumstances in which NUMA can be beneficial. Then, the findings of our study along with the drawn conclusions are amalgamated into the prototyping of a novel, dynamic, application-agnostic optimization mechanism for scaling up the performance of managed applications on NUMA systems.

This paper makes the following contributions:

- It conducts a study that characterizes the NUMA scalability of several Dacapo and Renaissance applications, and it classifies them into discrete categories.
- It proposes a novel, dynamic, and application-agnostic optimization mechanism for MREs. The proposed mechanism is lightweight as it capitalizes on the devised classification to: i) assess whether the performance of a running application can scale on a NUMA machine; and ii) adapt the execution of a running application based on the most suitable configuration.
- It presents a detailed performance evaluation of the proposed mechanism, showcasing that 26 out of 30 studied applications from Dacapo and Renaissance either gain speedup or avoid penalization. Additionally, the relative performance of the studied applications ranges from 0.66x up to 3.29x with a geometric mean of 1.11x, against a NUMA-agnostic execution.

2 Factors that Impact NUMA Performance

Several memory-related factors can impact the overall performance as well as the scalability of applications on NUMA systems [6, 10, 18]. One of those factors is the data dependency among the working threads of an application as it can potentially undermine the distribution of the application's workload across the NUMA system. Additionally, cache coherency is a key requirement for modern NUMA implementations that aim to provide a shared memory environment. The reason is that a program running on top of modern NUMA distributed architectures would tentatively place multiple replicas of its data in the cache memories which reside in the distributed nodes [23].

This section describes the MESIF protocol [11] (Section 2.1), a protocol introduced by Intel, and can be found in several commodity NUMA systems, as the one used in this work. Additionally, Sections 2.2 and 2.3 present two cases in which the data dependencies in the context of NUMA hinder data locality, and subsequently undermine scalability.

2.1 MESIF Cache Coherency Protocol

The MESIF protocol contains the following states that a cache line can be marked with:

- ***Modified***: the data is cached only on the current cache and its value has already been modified (dirty). Note that main memory should be firstly updated (write-back) upon a read request for the outdated main memory data. Then the *Modified* cache line transits to the *Shared* state.
- ***Exclusive***: the cached data matches the value in main memory (clean). It may transit to *Modified* upon a write request or to *Shared* upon a read request.
- ***Shared***: the cache line is clean but it is also present in other cache memories.
- ***Invalid***: the cache line is currently outdated (unused).
- ***Forwarding***: this state is a "special" *Shared* state, and it is used to mark that a node will be responsible for responding with the data upon a request. Considering that a *Shared* cache line is allowed to be present in multiple nodes, the MESIF protocol nominates one of those nodes to be responsible for responding with the data upon a request instead of the main memory [11].

Essentially, MESIF is a protocol that aims to take advantage of the fact that data can always be fetched faster from a cache memory than main memory - even from a remote node.

2.2 "Ping-Pong": Remote Write-After-Write

Ideally, multiple threads across NUMA nodes should process their "own" data without sharing any data with other threads; especially with those running on a remote NUMA node. However, the unfortunate case of an object that is being consecutively written by thread(s) from remote node(s) (we name this case as "Remote WAW") will not only increase the expensive memory accesses in the remote node, but also the invalidation of the cached data. Moreover, it will increase the pressure on the main memory, the interconnect traffic, and will contaminate the Last Level Cache (LLC). For instance, the data of a hot (consecutively written) Java object are written into the LLC of a NUMA node (home NUMA node). For each consecutive remote write access to the same object (Remote WAW), the following three steps will be performed: ❶ the data will be fetched to the LLC of the remote node (remote node access, interconnect traffic) which is now the new home node; ❷ the old home LLC will be invalidated (LLC pollution); and ❸ the data will be written back to main memory, thereby resulting in additional pressure to main memory.

2.3 "Write Conflicts": Remote Write-After-Read

As explained in Section 2.1, the *F* state of the MESIF protocol aims to mitigate contention for shared read data, because it allows a clean/unmodified cache line to be shared for reads among the nodes by providing a "copy" to the requesting node in order to avoid a main memory access [11]. However, *local* write accesses can still invalidate cached data across the system and retain contention. More specifically, in case a local write access is performed by a writing thread to an object that is read by remote node threads, the cache lines of the reader(s) will be invalidated. This will result in more LLC misses and remote node accesses, while it might also increase the main memory accesses. Moreover, the updated value is cached in the node of the writing thread, potentially by overwriting other cache lines, thereby resulting in more capacity cache misses.

3 Experimental Methodology

This section describes the platform (Section 3.1) that we used to prototype our research contributions, as well as the employed profiling tools. Additionally, Section 3.2 presents the characteristics of our experimental testbed. Finally, Section 3.3 gives an overview of the studied Java applications.

3.1 MaxineVM: A Research VM

Our characterization study and the proposed mechanism are performed in MaxineVM, a research VM written in and for Java. The modular design and ease in programming of this VM make it a suitable environment for research and experimentation. Additionally, MaxineVM provides multiple profiling tools, such as PerfUtil [20, 21] and NUMAProfiler [20], which can facilitate a study in the context of NUMA.

3.1.1 Java Profiling Tools for NUMA. PerfUtil [20, 21] is a MaxineVM profiler that interfaces with the Linux kernel's *perf* functionality and equips the VM itself with fine-grain utilization of the Hardware Performance Counters (HPC). It offers flexible utilization by enabling the monitoring of HPCs per-thread, per-core, or both. In addition, PerfUtil has the capability of time-multiplexing that allows for the concurrent measurement of a wide range of HPCs and operates with low overhead. PerfUtil is effective in the context of a NUMA system since it can be configured to monitor NUMA-related events (i.e., remote node accesses). Nevertheless, PerfUtil metrics lack correlation with application characteristics, such as data dependencies between application threads, despite being closely linked to overall performance observations. Scalability and performance in NUMA are impacted by higher-level factors, such as serial code segments, contention on shared resources, data locality and load balancing [23].

To capture metrics related to the aforementioned properties, we used NUMAProfiler. NUMAProfiler is a Java object profiler for MaxineVM that provides a higher-level perspective of object allocations and accesses per thread. NUMAProfiler can classify the shared object accesses, which can spotlight any inter-thread data dependencies. To facilitate the classification of the shared object accesses, MaxineVM stores the ID of the "owner" thread for each object in the misc word of the object header. If a thread that is not the "owner" thread accesses an object, the access is classified as shared. It is important to note that the concept of the "owner" thread is abstract and context-dependent. This work defines the *last writer* thread as the "owner" thread. In the context of a NUMA system, where the distribution of a workload across the system is crucial, this definition seems more reasonable, as it takes into account the effects of Remote WAW and Remote WAR dependencies, as discussed in Sections 2.2 and 2.3.

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

	Single Node	**Dual Node**	**All Nodes**
# of CPUs	1	2	2
# of Utilized Cores	8	8	16
LLC Size (MB)	20	40	40
Mem. Controllers	4	8	8
DRAM Size (GB)	192	384	384
Java Heap Size (GB)	100	100	100
HyperThreading	off	off	off
Page migration	off	off	off

3.2 Testbed: Hardware & Software Characteristics

Table 1 presents the characteristics of the testbed which is a 2-node NUMA machine. Additionally, it describes three running configurations: *Single Node*, *Dual Node*, *All Nodes*. The *Single Node* configuration models a Uniform Memory Access (UMA) machine as it deploys one NUMA node. On the other hand the *Dual Node* configuration models a Non-Uniform Memory Access (NUMA) machine. Note that both configurations use an equal number of cores and differ in the number of active NUMA nodes. A comparison between *Single Node* and *Dual Node* excludes potential scalability effects (Section 4.3), thereby simplifying the observation of the remote LLC regarding the impact in the memory behavior and data locality. Lastly, the *All Nodes* configuration deploys all system resources and is used in Section 4.4 (NUMA scalability assessment) and in Section 5 (optimization mechanism). To prevent additional performance and behavior variations, HyperThreading (HT) and Page Migration (PM) are disabled while the CPU frequency is fixed to 2.9 GHz via the ACPI CPU frequency driver. Note that, PM is not always beneficial due to the higher cost of a migration compared to the cost of a remote node access [21]. MaxineVM is deployed with the (default) SemiSpace GC while the OS operates with the default NUMA allocation policy (MPOL_DEFAULT).

3.3 Java Benchmarks Overview

A collection of 30 Dacapo [3] and Renaissance [24] applications is utilized for both the scalability study and the optimization mechanism evaluation. More specifically, we use the latest pre-built maintenance version of Dacapo (dacapo 9.12 MR1) [5], and the pre-built 0.11.0 release of Renaissance (https://github.com/renaissance-benchmarks/renaissance/tree/v0.11.0). The number of iterations to reach a steady state is selected following best practices for Dacapo [16] and Renaissance [24]. To ensure adequate steady state iterations, we add ten additional iterations beyond the recommended number for each benchmark. The largest input size is used along with eight threads, wherever possible. Note that the Dacapo applications allow the user to configure both the input size and the deployed threads with some exceptions (i.e. `avrora`) where the thread number is determined by the input size. On the contrary, the Renaissance benchmarks have a small and a default/large input size, while most of the benchmarks automatically deploy worker threads equal to the number of available cores.

4 NUMA Scalability Study

This section presents a characterization study of the Dacapo and Renaissance benchmark suites with respect to NUMA scalability. The performance of Java applications in a NUMA system can be penalized for reasons related to the remote memory hierarchy (i.e., exhaustive remote accesses, repetitive invalidation of cached data, etc. - Section 2). However, even compute-bound applications (such as `als` - Figure 1) can be penalized in a NUMA system. Therefore, a characterization that not only considers memory but also scalability properties is necessary. Towards that objective, several application properties are examined in our study, as follows: the degree of parallelism and balance of a workload (Section 4.1), the data dependencies of a workload (Section 4.2), and the data locality (Section 4.3).

4.1 Workload Parallelism & Balance

Table 2 presents a collection of metrics per application to evaluate workload parallelism and balance. The C and I metrics derive from PerfUtil, whereas the OA metric derives from NUMAProfiler. All metrics are obtained using the "Single Node" run configuration as the objective is to evaluate the application properties. The threads of each application are classified to "Workers" (those which process the workload), "Auxiliary" (non-worker threads i.e., timers, finalizers, etc.), and the "Main" thread. The workload of a thread is quantified as the number of hardware instructions it retires. The workload of each type is expressed as a percentage over the

Table 2. Workload Parallelism and Balance of applications.

Benchmarks	Main	Aux		Workers Imbalance			
	I %	#	I %	#	C [%]	I [%]	OA [%]
avrora	1	0	0	26	30	67	92
fop	100	1	0	0	0	0	0
h2	29	1	0	8	2	1	1
jython	100	3-4	0	0	0	0	0
luindex	85	1-2	15	0	0	0	0
lusearch	7	0	0	8	2	2	0
lusearch-fix	7	0	0	8	2	2	0
pmd	1	1	0	8	65	69	75
sunflow	0	10	0	8	2	2	1
xalan	0	0	0	8	0	1	0
akka-uct	0	12	0	184-200	72	118	44
reactors	5	3-4	0	8	26	28	46
als	1	80-84	1	4	6	6	12
chi-square	2	77-79	0	2	1	1	0
gauss-mix	2	75	0	2	1	0	0
log-regression	6	75-80	1	2	0	0	0
movie-lens	8	109-147	4	4-5	21	21	0
naive-bayes	1	75	0	9	25	25	30
db-shootout	0	2-3	0	48	75	88	79
fj-kmeans	1	1	0	25-412	106	109	135
future-genetic	0	1	0	10-12	50	48	87
mnemonics	100	1	0	0	0	0	0
par-mnemonics	22	1	0	7-8	265	265	226
scrabble	13	1	0	7	4	2	2
neo4j-analytics	0	27-28	0	4	68	67	60
rx-scrabble	3	2	0	8	149	153	222
dotty	100	1	0	0	0	0	0
scala-doku	100	1	0	0	0	0	0
scala-kmeans	100	1	0	0	0	0	0
philosophers	0	1	0	9	34	35	23
scala-stm-bench7	1	1	0	9	56	108	90

total retired hardware instructions. The workload carried out by worker threads is further analyzed under the scope of balance (Table 2 - Workers Imbalance). The variables C, I, and OA refer to the imbalance of CPU cycles, retired hardware instructions, and object accesses respectively, between the worker threads. For example, the worker threads of `avrora` show 30% imbalance in CPU cycles (C), 67% in hardware instructions (I), and 92% in object accesses (OA). To assess the balance of a variable X (i.e., the retired hardware instructions) between N threads, the Equation (1) is used. The Equation (1) calculates the **imbalance in X** between the N threads [6]. A high number of standard deviations can yield in a high imbalance, assuming that the average value is constant (Imbalance = 0% means "totally balanced").

$$Imbalance\ in\ X = \frac{stdev(X_{thread1}, ..., X_{threadN})}{average(X_{thread1}, ..., X_{threadN})} \quad (1)$$

An assessment of the memory contribution to the workload imbalance is necessary because the imbalance of hardware instructions does not sufficiently reflect the overall imbalance of the workload (due to variations in complexity and latency among hardware instructions i.e., arithmetic vs memory instructions). Therefore, the imbalance of object accesses is examined as it effectively reflects the computations-versus-memory heterogeneity of the worker threads. Hereafter, we

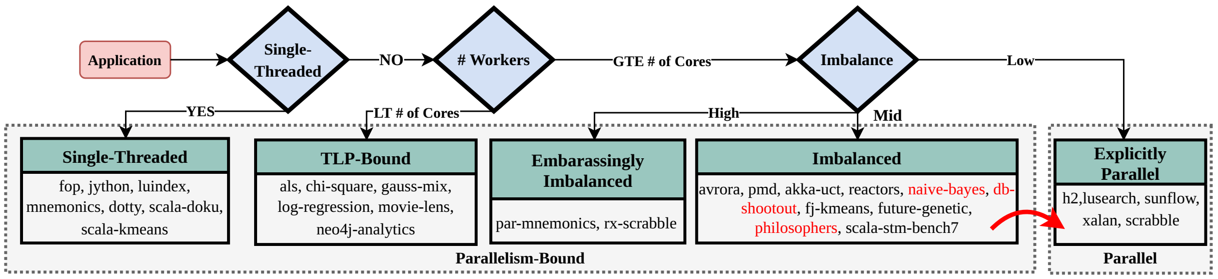
Application
Single-Threaded
NO
# Workers
GTE # of Cores
Imbalance
Low
YES
LT # of Cores
High
Mid
Single-Threaded
fop, jython, luindex, mnemonics, dotty, scala-doku, scala-kmeans
TLP-Bound
als, chi-square, gauss-mix, log-regression, movie-lens, neo4j-analytics
Embarassingly Imbalanced
par-mnemonics, rx-scrabble
Imbalanced
avrora, pmd, akka-uct, reactors, naive-bayes, db-shootout, fj-kmeans, future-genetic, philosophers, scala-stm-bench7
Explicitly Parallel
h2,lusearch, sunflow, xalan, scrabble
Parallelism-Bound
Parallel

Figure 2. NUMA Scalability Characterization: Workload Parallelism & Balance.

will use the term "computation-memory heterogeneity" to refer to the computations-versus-memory heterogeneity. Nevertheless, the imbalance of hardware instructions and object accesses might not imply that the workload is performance-wise imbalanced, because the memory-derived stalls might offset the observed computations-memory heterogeneity in terms of CPU cycles. Therefore, we also examine the impact of this heterogeneity on performance through the imbalance of an application in terms of the CPU cycles.

Considering the aforementioned metrics, we can classify the applications in five discrete categories ("Single-Threaded", "TLP-Bound", "Embarrassingly Imbalanced", "Imbalanced", and "Explicitly Parallel"). Figure 2 illustrates a flow chart of the aforementioned categories along with the examined metrics. All the categories, but the "Explicitly Parallel", indicate that the applications do not exhibit the degree of parallelism that is needed to efficiently scale on a NUMA system; hence are very likely to be parallelism-bound.

4.1.1 Single-Threaded:

This category includes applications that do not spawn parallel threads, thereby being unable to scale. The workload of those applications is either driven exclusively by the main VM thread (`fop` and `luindex`) or by up to two auxiliary threads, such as a `Finalizer` (`dotty`), and a `LogManagerCleaner` (`jython`, `mnemonics`, `scala-doku`, and `scala-kmeans`).

4.1.2 TLP-Bound:

In case an application deploys fewer threads than the available CPU cores, its scalability is bound because the exhibited Thread Level Parallelism (TLP) is below the capacity of the system. Among the studied applications, those that are classified as "TLP-Bound" spawn 2-4 worker threads. Note that the imbalance in hardware instructions of `als` and `movie-lens` is directly reflected to imbalance in CPU cycles (`als`: C=6%, I=6%, OA=12%, `movie-lens`: C=21%, I=21%, OA=0%). Such a fact indicates that the memory operations do not significantly affect performance, thereby revealing the compute-bound nature of those applications.

4.1.3 Embarrassingly Imbalanced:

Applications that belong to that category show extreme imbalance in hardware instructions. This fact is justified by the observation that only one worker thread carries out the 80% of the overall workload (even though that 7-8 workers are spawned). This observation was revealed by examining the total retired hardware instructions per worker thread. Additionally, the imbalance in object accesses indicates that the workers of those applications are imbalanced also in terms of memory. The imbalance of CPU cycles cross-validates that the worker threads are indeed imbalanced, and they follow the asymmetric trends of hardware instructions and object accesses.

4.1.4 Imbalanced:

Considerable but lower imbalance in hardware instructions is also observed in other parallel applications. There is a strong positive linear correlation (0.8) between the imbalance in hardware instructions and the imbalance in CPU cycles even though there are two outliers: `avrora` and `scala-stm-bench7`. However, the correlation is lower (0.66) between the imbalance in CPU cycles and the imbalance in object accesses. This denotes that the effect of memory in the imbalance of CPU cycles varies across the applications. To assess whether the observed computational and/or memory imbalance is harmful, we have inspected some additional characteristics of the applications.

`Avrora` is composed of 11 individual workloads that are processed in parallel. The first 4 workloads utilize 7, 3, 7 and 2 threads respectively, while the remaining 7 deploy only one thread; hence, they are single-threaded (26 worker threads in total). This discrepancy between the different workloads can explain the observed computation-memory heterogeneity of worker threads. The imbalance of CPU cycles which is lower than the imbalance of hardware instructions and object accesses, denotes that the computation-memory heterogeneity has little effect on performance probably due to good memory locality (~1% LLC Miss Rate in "Single Node").

`Pmd` analyzes multiple source code files in an imbalanced manner due to the unequal sizes of the input files [7]. This fact probably causes the observed computation-memory heterogeneity of worker threads. In contrast to `avrora`, the computation-memory heterogeneity of worker threads in `pmd` is directly reflected in the imbalance of CPU cycles. Moreover, it is noteworthy that the work-stealing strategy that `pmd` deploys to maintain workload balance, fails to effectively counterbalance small and large jobs.

`Akka-uct` implements the Unbalanced Cobwebbed Tree (UCT) algorithm [28] in the Akka actors framework. UCT

Table 3. Summary of the Root Causes of Imbalance.

Root cause	Applications
Non-uniform workload distribution across the deployed workers	`avrora`, `pmd`, `akka-uct`, `fj-kmeans`
Dominant single-threaded/serial algorithm/code sections along with minor explicitly parallel sections	`reactors`, `scala-stm-bench7`
Synthetic incorporation of different workloads	`avrora`, `reactors` `db-shootout`
Outlier special-cause threads among balanced workers	`naive-bayes`, `philosophers`

processes a tree of tasks with variable size which are assigned to workers by the Akka dispatchers via a shared task queue structure [28]. Rosa et al. [25] report non-uniform task distribution across actors which is also confirmed by the imbalance in hardware instructions of the worker threads (Table 2). Lower imbalance in object accesses (compared to hardware instructions) seems to counterbalance performance, thereby leading to milder imbalance in CPU cycles.

`Reactors` incorporates ten individual message passing Savina [12] benchmarks that are implemented into the Reactors.IO framework [13]. They are of diverse message passing counts, processed sequentially by an eight-worker fork-join pool and all are single-threaded but two. This workload discrepancy can explain the observed imbalance of object accesses; however, this gap is diminished in respect to hardware instructions and CPU cycles imbalance.

The workload of `naive-bayes` is equally distributed to eight out of the nine deployed workers. Therefore, one thread has a different role than the rest. However, the workload is dominated by the eight homogeneous workers because the imbalance is maintained in low levels. Therefore, this application is an exception and fits better to the "Explicitly Parallel" category. `Philosophers` includes one special thread ("camera") along with eight balanced workers. This diversity in the roles of the deployed threads can explain the observed imbalance similarly to the `naive-bayes` case; hence, this application is also classified as "Explicitly Parallel".

`Db-shootout` incorporates three synthetic `Lmdb` workloads. They are implemented in the `MapDB`, `ChronicleMap`, and `MvStore` frameworks and are executed sequentially. Although each workload deploys eight workers and performs the same amount of DB operations (500k reads + 500k writes), they differ regarding the amount of object accesses that each one performs. This results in the observed imbalance of hardware instructions and object accesses because each sub-workload itself is quite balanced (`MapDB` - 5%, `ChronicleMap` - 25%, `MvStore` 36%). Consequently, the observed imbalance of `db-shootout` is illusional, hence this application fits better to the "Explicitly Parallel" category.

`Fj-kmeans` implements the k-means algorithm using an 8-worker Fork-Join thread pool. The observed imbalance in CPU cycles that is also reported by Rosales et al. [26] probably denotes either computation-memory heterogeneous subtasks or/and sub-tasks of unequal size. The imbalance in hardware instructions and object accesses in Table 2 indicate both.

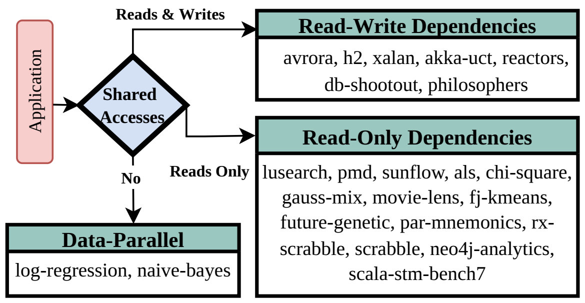
Application
Shared Accesses
Reads & Writes
Reads Only
No
Read-Write Dependencies
avrora, h2, xalan, akka-uct, reactors, db-shootout, philosophers
Read-Only Dependencies
lusearch, pmd, sunflow, als, chi-square, gauss-mix, movie-lens, fj-kmeans, future-genetic, par-mnemonics, rx-scrabble, scrabble, neo4j-analytics, scala-stm-bench7
Data-Parallel
log-regression, naive-bayes

Figure 3. NUMA Characterization: Data Dependencies.

Sub-tasks of unequal size and the ineffectiveness of the work stealing in preserving the balance are rather counterintuitive findings for a Fork-Join application. `Scala-stm-bench7` is single-threaded for ~80% of the execution time. Eight balanced workers are deployed only for the remaining 20% of the total execution time. This explains the low imbalance of CPU cycles in comparison to the hardware instructions. Consequently, this application can be split into two phases: a) single-threaded alike, and b) explicitly parallel.

The analysis of the "Imbalanced" applications above reveals that the observed imbalance arises from multiple underlying reasons and root causes (summarized in Table 3). It should be noted that the last two root causes may suggest that an application is not actually "Imbalanced".

4.1.5 Explicitly Parallel: Applications that deploy multiple (equal or greater than the available CPU cores) and balanced workers are considered as "Explicitly Parallel". Even though all the deployed worker threads of `h2` are balanced, almost 30% of its workload is carried out by the main thread. Such a fact indicates that `h2` has extensively serial phases.

4.2 Data Dependencies

This section discusses the data dependencies between the worker threads by examining the metric of shared object accesses via NUMAProfiler (Section 3.1.1). The measurements are performed in the "Single Node" run configuration because this metric is not affected by the underlying hardware. A high number of shared reads or writes is a strong indication that an application contains shared resources. These resources, particularly if they contain Remote WAW and WAR dependencies (Section 2.2 and Section 2.3), can impede the distribution of a workload across the NUMA system. Table 4 lists the percentage of shared accesses over total object accesses. Three major categories are observed: "Data Parallel (DP)", "Read & Write Dependencies (RWD)", and "Read-Only Dependencies (RD)". The following paragraphs discuss the classification that is illustrated in Figure 3.

4.2.1 Data-Parallel: Having neither considerable shared reads nor shared writes, this category denotes that it probably is free of data dependencies. The absence of shared data

Table 4. Shared Accesses. "Owner" = Last Writer.

TLP Bound			Embarrassingly Imbalanced			Imbalanced			Explicitly Parallel		
	Sh R	Sh W		Sh R	Sh W		Sh R	Sh W		Sh R	Sh W
als	7.7%	0.1%	par-mnemonics	47.9%	0%	avrora	62.3%	2.5%	h2	36.5%	1.7%
chi-square	27.1%	0%	rx-scrabble	48.2%	0%	pmd	34.9%	0.2%	lusearch	22.8%	0.2%
gauss-mix	35.8%	0%				akka-uct	67.2%	0.9%	sunflow	83.8%	0%
log-regression	1.7%	0.2%				reactors	57.1%	5.3%	xalan	15.9%	3.1%
novie-lens	29.9%	0.4%				fj-kmeans	59%	0.1%	naive-bayes	3.2%	0.2%
neo4j-analytics	38.1%	0%				future-genetic	35.3%	0.9%	db-shootout	22.9%	2.0%
						stm-bench7	35%	0%	scrabble	47.2%	0%
									philosophers	51.5%	2.5%

among threads is a sufficient condition for maintaining locality of data even if threads are naively scheduled to run across NUMA nodes. The scalability of those applications on a NUMA system is not affected by data dependencies.

4.2.2 Read & Write Dependencies: Those applications have a considerable amount of shared read and shared write accesses. Such a fact indicates strong dependencies among the working data. Objects updated by multiple worker threads in a NUMA system harm locality in case those threads are scheduled on different NUMA nodes (Sections 2.2 and 2.3). Such a phenomenon inevitably increases the pressure on LLC/Memory, and the interconnect traffic. Consequently, this kind of data dependencies is very likely to prevent those applications from scaling.

4.2.3 Read-Only Dependencies: A considerable amount of shared reads with negligible or even zero shared writes indicates the existence of shared data. However, such a fact might not lead to increased pressure on the LLC/memory and the interconnect; thus, might not prevent scalability (i.e., `sunflow`). The impact of "Read-Only Dependencies" is tightly related to the amount of "write conflicts" (Section 2.3).

4.3 Data Locality

This section assesses the data locality of the applications by comparing the LLC miss rate of the "Dual Node" configuration with the equivalent of "Single Node". As explained in Section 3.2, this comparison aims to reveal how the memory behavior of an application is affected by the remote LLC and memory. Table 5 presents the Read (R) and Write (W) LLC Miss Rate (%) per application. The metrics are obtained via PerfUtil using the "Single Node" (S) and "Dual Node" (D) configurations (excluding the "Single-Threaded" applications). The objective is to highlight the effect of NUMA on the data locality by observing the difference in the LLC Miss Rate between the configurations. In addition, Figure 4 illustrates the percentage of LLC misses in Read (grey) & Write (black) that are served by the main memory of a remote node. The metrics reported in this figure are obtained with the "Dual Node" configuration, and they indicate whether the observed difference in the LLC Miss rate is related to "write-conflicts" (Section 2.3). The following paragraphs discuss data locality

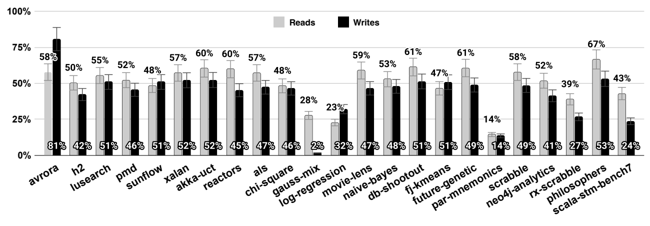

Figure 4. Read & Write LLC Misses of a Remote Node.

in association with the categories introduced in Sections 4.1 and 4.2, with the exclusion of the "Single-Threaded" category.

4.3.1 TLP-Bound: The data locality of these applications is negligibly affected by NUMA. This is because the "TLP-Bound" applications usually deploy few workers, and consequently the Linux kernel that aims to maintain the locality of data is very likely to reside the threads and the working data in the same NUMA node. Nevertheless, some counterintuitive cases exist. For example, `als` and `movie-lens` present a considerable increase (+13%, +5% accordingly) in the observed LLC read miss rate and `neo4j-analytics` presents a slight decrease in LLC read misses (-1%). Those three applications deploy four workers each, along with multiple auxiliary threads of the Spark and Neo4j engines respectively (Table 2). As a result, this amount of threads can force the OS to spread the threads in multiple NUMA nodes. The percentage of remote memory reads that those applications show is the highest among the "TLP-bound" (57%, 59%, and 52%) and indicates that the worker threads were indeed scheduled to run on both NUMA nodes of the system. In addition, the "Read-Only Dependencies" that these applications have (Section 4.2) imply that the spread threads communicate and share data. However, those applications are differentiated regarding the data locality from Single to Dual Node. The fact that the the high number of remote memory reads is accompanied by more LLC read misses in Dual Node for `als` and `movie-lens` but not for `neo4j-analytics` implies that the latter does not suffer from "write-conflicts", while `als` and `movie-lens` do. More specifically, the `als` and `movie-lens` apparently have objects that are repetitively read by threads that run on a remote node, written/updated by the owner thread and inevitably lead to more LLC misses. On the other

Table 5. LLC Miss Rates per "Workload Parallelism & Balance" and "Data Dependencies" Classification.

	TLP Bound	R		W		**Embarrassingly Imbalanced**	R		W		**Imbalanced**	R		W		**Explicitly Parallel**	R		W	
		S	D	S	D		S	D	S	D		S	D	S	D		S	D	S	D
RD	als	6	19	44	50	par-mnemonics	8	9	55	55	pmd	2	2	58	56	lusearch	0	2	54	56
	chi-square	10	13	60	59	rx-scrabble	4	7	59	61	fj-kmeans	70	71	44	63	sunflow	2	2	64	62
	gauss-mix	3	3	60	60						future-genetic	0	14	48	58	scrabble	2	22	33	72
	movie-lens	4	9	51	54						stm-bench7	16	31	68	63					
	neo4j-analytics	24	23	63	61															
RWD											avrora	0	9	6	34	h2	35	36	48	49
											akka-uct	21	32	59	64	xalan	0	4	47	51
											reactors	3	23	29	47	db-shootout	10	29	60	64
																philosophers	0	21	28	50
DP	log-regression	54	56	57	57											naive-bayes	17	22	59	58

hand, `neo4j-analytics` does not suffer from that effect. This observation confirms the intuitive expectation that the "Read-Only Dependencies" can behave either like "Data Parallel" (as in the case of `neo4j-analytics`) or like having "Read & Write Dependencies" (as in the case of `als`, and `movie-lens`) due to the effect of "write conflicts". Moreover, `als` and `movie-lens` show the highest performance degradation among all "TLP-Bound" (Figure 1). In particular, `als` is a compute-bound application [24] which turns out to be memory-bound when run in a NUMA system due to the data dependencies that harm locality.

4.3.2 Embarrassingly Imbalanced:

The negligible difference in the LLC Miss Rate and the low number of remote node accesses reveal the main characteristics of these applications. The multiple workers that an "Embarrassingly Imbalanced" application spawns are not concurrently active, hence it essentially behaves similarly to a "Single-Threaded" application. As a result, the OS settles up the application on only one NUMA node and no significant difference is observed in the locality of data and performance. Therefore, the offering of multiple NUMA nodes in such an application even though it is not harmful, it is not likely to be beneficial.

4.3.3 Imbalanced:

`Future-genetic` and `scala-stm-bench7` show a considerable increase in the LLC Miss Rate of read operations, probably due to the number of "write conflicts" over the shared read objects (similarly to `als` and `movie-lens`). These applications deploy enough workers in order to force the OS to schedule them across the NUMA nodes but contain only read dependencies.

`Pmd` is not bound by data locality since its LLC Miss Rate is not affected. Such an observation is expected considering that each worker processes an individual file. Note that, `pmd` is one more example of an application with "Read-Only Dependencies" which behaves as being "Data Parallel".

The LLC Write Miss Rate of `fj-kmeans` is heavily affected in Dual Node, but data locality of Reads is not. However, the LLC Read Miss Rate of `fj-kmeans` is already high (70%) even in the Single Node configuration. This fact indicates that locality is harmed due to limited LLC capacity and/or irregular memory access patterns which is attributed to the application implementation. More specifically, the `fj-kmeans` recursively splits the working data (data points) into smaller chunks until a chunk of desired size is (randomly) assigned to the first available worker from a Fork/Join thread pool in order to perform the centroid calculations. Therefore, each worker processes non "neighboring" data chunks, hence it cannot benefit from spatial locality and hardware prefetching. This essentially is an irregular data pattern because a worker can process data chunks from any segment of the working dataset. This practice can explain the very high LLC Read Miss Rate, even in Single Node. The increase in LLC Read Miss Rate for Dual Node is avoided though due to the read-only nature of this phase. Moreover, the k-means algorithm updates the calculated centroids after processing all forked subtasks. Therefore, the LLC write miss rate increase in Dual Node (+19%) can be attributed to the impact of the "ping-pong" effect on this update phase (that the workers are spread in both NUMA nodes and repetitively invalidate already cached data which is about to be written/updated). As a result, it is clear that the data locality bounds the scalability of `fj-kmeans` on a NUMA machine.

The "Imbalanced" applications with "Read & Write Dependencies" (`avrora`, `akka-uct`, `reactors`) show high increase in LLC Miss Rate for both reads and writes. Such an observation is expected because those applications deploy enough workers in order to force the OS scheduler to spread them in all NUMA nodes, while also they contain strong read and write dependencies between the workers. For example, the Miss Rates of `avrora` in the Single Node configuration (R = 0.1%, W = 6%) imply that this application has good data locality and does not suffer from capacity misses even in the smaller LLC of the Single Node. As a result, the observed increase of the LLC Miss Rates in Dual Node is attributed only to the data dependencies between the workers that run in both NUMA nodes. This conclusion is also supported by the number of remote node memory accesses (R = 58%, W = 81%) and the increase in LLC Miss Rates (R: +9%, W: +28%). It becomes apparent that spreading the workers of this application across NUMA nodes without considering the data dependencies breaks the locality by invalidating the already cached data, and consequently leads to more LLC Misses. The `akka-uct` application seems to be affected mostly by write-conflicts over shared read data (similarly to `future-genetic` and `scala-stm-bench7`). Finally, the `reactors` application has the highest percentage of shared write object accesses (5.3%, see Table 4) among all applications. The shared writes imply that the workers write/update objects that are owned by other workers. In case the writer resides in a different NUMA node than the object, a remote node access occurs, the data is updated and transferred in the LLC of the node of the writer, and the cached data (if any) of any other node is invalidated ("ping-pong" effect, see Section 2.2). Therefore, this situation will lead to additional accesses to the remote nodes upon the occurrence of any read or write operation by a remote node worker. Moreover, it will lead to more cache misses because the transfer will overwrite already cached data. The latter is confirmed by the observed increase in LLC Miss Rates (R: +20%, W: +18%). Consequently, the shared write dependencies seem to harm data locality of `reactors`.

4.3.4 Explicitly Parallel: The data locality of `db-shootout` and `philosophers` seems to be affected the most among the "Explicitly Parallel" applications with "Read & Write Dependencies". The LLC Miss Rates of `philosophers` for reads and writes are significantly increased from Single to Dual Node (R: +21%, W: +22%). However, this application has a considerable amount of shared writes (2.5%) and the implemented dining philosophers algorithm is a well known synchronization problem over shared resources [27]. Consequently, the lack of data locality in Dual Node for this application is attributed to the data dependencies. On the contrary, `h2` and `xalan` maintain locality in Dual Node, thereby indicating that the observed data dependencies probably do not concern the same objects. The data locality of the applications with "Read-Only Dependencies" is less likely to be harmed in NUMA compared to those that have "Read & Write Dependencies"; however, there are such cases. For instance, `scrabble` shows considerable increase in the LLC Miss Rate (R: +20%, W: +38%), denoting clearly that its locality is heavily harmed by NUMA. In addition, after inspecting the Branch Prediction Unit Misses per Kilo Instructions (BPU MPKI), it turns out that `scrabble` has one of the highest BPU MPKI (scrabble = 4.11, geomean = 1.35). Such a fact along with the observed LLC Miss Rate increase in Dual node is very likely to reflect irregularities in the memory access pattern. `Scrabble` is structured around a centralized `HashSet` which acts as a reference dictionary for the *scrabble* game. Consecutive read and/or write operations on a HashSet can create irregular memory access patterns because two semantically neighbor buckets do not necessarily neighbor into the HashSet. Moreover, bucket manipulation does not require serial traversal of the data structure, hence a typical HashSet cannot benefit from cache and prefetching mechanisms. As a result, `scrabble` is very likely to get its buckets accessed in a random order. To verify this assumption, we replaced the `HashSet` data structure with an `ArrayList` and observed mild increase in LLC Read Miss Rate (S:14%, D:19%) and decrease in LLC Write Misses (S:69%, D:63%). Even though the `ArrayList` is a locality friendly data structure, it is more resource-consuming than the `HashSet`. For example, the `ArrayList` version of `scrabble` executes 124x more instructions and 120x more L1 accesses than the `HashSet` version, thereby leading to a trade-of between data locality and performance. Consequently, it is clear that `scrabble` suffers from irregular memory access patterns that are related to the HashSet. As a result, the irregularity in memory access patterns heavily damages data locality and can also explain the heavy performance degradation observed in Figure 1. `Naive-bayes` is the only application that belongs to the "Explicitly Parallel" and "Data Parallel" categories. The data locality of this application is negligibly affected which is expected because it belongs to a category without data dependencies. The only potential menace regarding data locality for applications that belong to these categories ("Explicitly Parallel" and "Data Parallel") is the irregular memory access pattern. `Naive-bayes` does not exhibit such a pattern since no extreme increase in LLC Miss Rate is observed. However, such a corner-case application would be an interesting addition to the benchmark suites.

4.4 Discussion on Performance Scalability

To attest the findings of the characterization study as discussed in previous paragraphs, we measured the performance of each application running with two different configurations, "All Nodes" against "Single Node" (Table 1). Table 6 presents the relative performance of the execution with the "All Nodes" configuration against the "Single Node" configuration. The most scalable applications belong to the "Explicitly Parallel" category since they exhibit no shared writes (RD or DP). Another observation is the significant performance degradation of the "Locality-Bound" applications (i.e., `fj-kmeans`, `scrabble`) which are illustrated using a grey background. None of the "Imbalanced" applications can scale its performance, thereby validating that data dependencies in combination with the lack of workload parallelism and balance can impact scalability. Furthermore, the observed performance of the "TLP-Bound" and "Embarrassingly Imbalanced" applications confirms that the best-case scenario for these categories is only to avoid any performance loss. However, this scenario can be achieved only if there are no data dependencies, and consequently, it might be more efficient to run those applications on one NUMA Node.

Table 6. NUMA Characterization of Dacapo & Renaissance.

	TLP Bound	Perf.	Embarrassingly Imbalanced	Perf.	Imbalanced	Perf.	Explicitly Parallel	Perf.	
RD	als	0.87x	par-mnemonics	0.89x	pmd	0.96x	lusearch	1.29x	Locality Bound
	chi-square	1.01x	rx-scrabble	0.98x	fj-kmeans	0.64x	sunflow	1.87x	
	gauss-mix	1.01x			future-genetic	0.83x	scrabble	0.39x	
	movie-lens	0.86x			stm-bench7	0.90x			
	neo4j-analytics	0.93x							
RWD					avrora	0.86x	h2	0.83x	
					akka-uct	0.85x	xalan	1.62x	
					reactors	0.79x	db-shootout	0.73x	
							philosophers	0.59x	
DP	log-regression	1.00x					naive-bayes	1.86x	

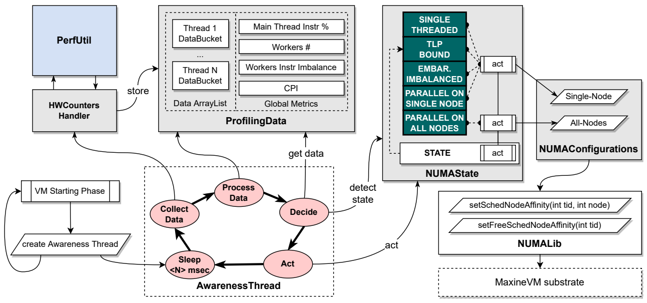

Figure 5. Overview of the Optimization Mechanism.

5 Optimization Mechanism

The previous section analyzed our characterization study that resulted in classifying several Dacapo and Renaissance applications into four categories. However, to perform the classification we had to employ two profiling tools and observe several low-level and high-level metrics using NUMAProfiler and PerfUtil, respectively. Such an approach is not transferable in the context of an online optimization due to the high overhead of the detailed profiling.

Therefore, we designed a novel mechanism that is prototyped in MaxineVM to detect whether the performance of JVM applications can scale on NUMA systems, and adapt their execution configuration online. The proposed mechanism utilizes exclusively the low-overhead profiling capabilities of PerfUtil for a running application, and it capitalizes on the findings of our characterization study (Section 4) to apply the most suitable running configuration. In particular, Section 5.1 presents the design and implementation of the proposed mechanism. Section 5.2 discusses the overhead of the proposed mechanism, whereas Section 5.3 analyzes the end-to-end performance of the studied applications.

5.1 Design and Implementation

MaxineVM is equipped with an optimization mechanism (`-XX:+NUMAOpts` option) that can dynamically and iteratively profile and classify a running application, and finally applies the proper run configuration. The mechanism is implemented as a new daemon VM-internal background thread (named "Awareness Thread") that implements all the actions performed within the mechanism, as shown in Figure 5. All actions in the workflow are depicted in red, while the main components of the mechanism are depicted in grey. The "Awareness Thread" is initialized along with the other VM-internal threads (i.e., main, VmOperations, etc.) during the starting phase of MaxineVM. Once it is initialized, the "Awareness Thread" becomes a daemon, and iterates over a set of actions (Sleep, Collect Data, Process Data, Decide, Act); each of those encloses a fragment of logic of the mechanism. The "Awareness Thread" dynamically coordinates the process of profiling and decision-making; hence it does not involve the application threads to additional duties. Any iteration of the "Awareness Thread" starts after a sleep time interval which regulates the operation frequency of the mechanism and subsequently how often a new decision is taken. The time interval is set to 200ms after experimentation. The "Collect Data" action obtains and stores the profiling data into buffers. Thereafter, the raw profiling data are processed during the "Process Data" action in order to calculate the following metrics: the percentage of main thread instructions over total retired instructions, the number of worker threads, the worker threads instructions imbalance, and the Cycles per Instruction (CPI). The aforementioned metrics are used to decide the application state as well as the most suitable running configuration for the application.

The mechanism is implemented to operate on a separate thread to offload any extra duties from the application threads. However, an increase of the number of context switches is expected due to the deployment of an additional thread. Section 5.2 analyzes the overhead of the mechanism.

5.1.1 Decision-Making Logic. The goal of the "Decide" action is to decide in which state the application currently is. An application is considered as `SINGLE_THREADED` if the main thread significantly dominates the retired hardware instructions (>80%). If it deploys equal or fewer workers than the number of available cores it is considered as `TLP_BOUND`. Additionally, an application is attributed as `EMBARRASSINGLY_IMBALANCED` if its imbalance in hardware instructions of worker threads exceeds 90%. In case an application has not fallen into one of the aforementioned categories, it is considered as "Parallel" ("Imbalanced" or "Explicitly Parallel"). However, it is not certain whether a "Parallel" application would be benefited by a NUMA system. The mechanism takes such a decision based on the following conservative strategy: Firstly, the application is temporarily forced to remain on a single NUMA node for one time interval. At the next interval the application is forced to run on all NUMA nodes. After those two time intervals, the mechanism has measured the `singleNodeCPI` and the `allNodeCPI`. The frequency of the system is fixed, therefore the CPI value is directly related to performance. Consequently, a reasonable decision can be taken by comparing the two CPI values. The application is classified as `PARALLEL_ON_ALL_NODES`, and therefore, it is allowed to continue running on all NUMA nodes if the `allNodeCPI` is lower (better) or equal to the `singleNodeCPI + cpiMargin`. Otherwise, the application is classified as `PARALLEL_ON_SINGLE_NODE`, thus it should settle back to a single node.

To eliminate the frequency based on which the mechanism takes a decision, we introduced a dynamic "stabilization" strategy. This strategy works, as follows. As long as the mechanism takes the same decision for a "Parallel" application, the optimization interval is increased by 200ms. The increase in the optimization interval aims to stabilize the decision mechanism and avoid unnecessary interruptions and ineffective migrations. Essentially, the "stabilization" strategy is based on a heuristic that assumes that a repetitive decision is less likely to be a coincidence. In case the decision of the mechanism alters at any point, the optimization interval is adapted to reset to its initial value (200ms).

Finally, the "Act" action is responsible for applying the proper run configuration according to a classification. The `PARALLEL_ON_ALL_NODES` classification leads to the "All-Nodes" configuration, while the rest lead to the "Single-Node" one. Note that, the aforementioned workflow was designed with the aim of being easily extensible towards further and more complex NUMA configurations.

5.2 The Overhead of the Mechanism

To evaluate the overhead of the proposed mechanism, we employed two builds that execute the applications on one node; `MaxineVM_vanilla` and `MaxineVM_fakeOpts`. `MaxineVM_vanilla` is an unmodified build of MaxineVM which is used as the baseline, whereas `MaxineVM_fakeOpts` is a MaxineVM build with the mechanism modified to always deploy the "Single Node" configuration (even in case the application is classified as `PARALLEL_ON_ALL_NODES`). We have measured and compared the average execution time of five runs with both builds for all applications. `MaxineVM_vanilla` in the "Single Node" configuration, `MaxineVM_fakeOpts` in the "All Nodes" configuration, while the applications are configured to deploy 16 threads, wherever possible. This comparison reveals the overall overhead of the proposed mechanism because `MaxineVM_fakeOpts` eliminates all NUMA-related factors that cause performance variations, and its performance is solely affected by the mechanism itself. Our experiments show that the geometric mean of the overall overhead for all studied applications is 0.9%.

5.3 Performance Evaluation

To measure the efficiency of the applications running with the optimization mechanism, we have used the `MaxineVM_vanilla` build with the "All Nodes" configuration as the baseline. Table 7 presents the relative performance of the execution with the optimization mechanism against the baseline execution for all applications. The reported relative performance is calculated using the average of five executions. As shown in Table 7, the proposed

Table 7. Relative Performance of the Optimization Mechanism against `MaxineVM_vanilla`.

Application	Speedup	Application	Speedup
scrabble	3.29x	chi-square	1.02x
f-kmeans	2.55x	log-regression	1.01x
philosophers	1.38x	luindex	1.01x
reactors	1.24x	naive-bayes	1.01x
akka-uct	1.16x	dotty	1.01x
future-genetic	1.15x	scala-doku	1.01x
als	1.15x	scala-kmeans	1.01x
db-shootout	1.14x	neo4j-analytics	1.00x
par-mnemonics	1.13x	mnemonics	1.00x
movie-lens	1.11x	jython	1.00x
avrora	1.07x	fop	1.00x
rx-scrabble	1.03x	sunflow	0.98x
pmd	1.03x	lusearch	0.97x
h2	1.02x	scala-stm-bench7	0.86x
gauss-mix	1.02x	xalan	0.66x
Geomean 1.11x			

mechanism improves performance by 11% using the geometric mean of all applications. Overall, some applications (i.e., `scrabble`, `fj-kmeans`, `reactors`, `db-shootout` and more) exploit the optimization mechanism, while others (i.e., `xalan` and `scala-stm-bench7`) are penalized by the mechanism. The most benefited applications are those which do not scale on a NUMA system, such as `scrabble` and `fj-kmeans`. Both benchmarks have been classified as "Locality Bound" in Table 6, hence they cannot scale on a NUMA system. In this case, both applications exploit the fact that the proposed mechanism has decided to deploy them on a single node. Therefore, the results confirm that the optimization mechanism can detect and prevent such a performance degradation.

An additional finding is the impact of the adaptive nature of the mechanism that is achieved through the dynamic "stabilization" strategy (Section 5.1.1). To evaluate the impact of this strategy, we have counted the number of "migrations" which corresponds to how many times the mechanism alters its previous decision/classification ("*SingleNode*" $\rightarrow$ "*AllNodes*", or "*AllNodes*" $\rightarrow$ "*SingleNode*"). The number of migrations is then compared against a non-adaptive (NA) version of the mechanism along with the impact in performance (Table 8). The "stabilization" strategy significantly benefits the `naive-bayes` (+0.35x) and `sunflow` (+0.26x) applications. It does not boost the performance of the penalized applications (i.e., `lusearch` and `scala-stm-bench7`), while it further penalizes `xalan` (-0.15x).

The variation in performance is also reflected in the number of migrations. More specifically, the "stabilization" strategy reduces the migrations for the benefited `naive-bayes` and `sunflow` applications (from 42 to 1, and from 31 to 7 respectively), while it has no significant impact in the migrations of the unaffected `lusearch` and `scala-stm-bench7`.

Table 8. Impact of the "stabilization" Strategy in Performance and the number of Migrations.

Application	Performance	Migrations	
	Speedup	NA	Stable
avrora	-0.05x	35	32
pmd	0x	0	0
akka-uct	+0.01x	192	122
reactors	+0.02x	0	0
fj-kmeans	0x	3	2
future-genetic	0x	718	689
scala-stm-bench7	-0.01x	71	72
h2	-0.01x	115	24
lusearch	+0.01x	30	22
sunflow	+0.26x	31	7
xalan	-0.15x	28	3
naive-bayes	+0.35x	42	1
db-shootout	-0.03x	177	21
scrabble	-0.01x	62	3
philosophers	-0.04x	41	3
GEOMEAN	**+0.03x**		

However, the migrations are also reduced for the penalized `xalan` (from 28 to 3) which is counterintuitive.

These observations lead to the following conclusion. The scalability of `naive-bayes` and `sunflow` is penalized by the frequent migrations. By reducing the frequency that the mechanism operates, the migrations are avoided. Hence, those applications can benefit from the deployment on two NUMA nodes. Nevertheless, this decision does not apply to all applications. A cause for that differentiation can be the fact that some applications do not have uniform behavior (i.e., their algorithm has multiple and different phases). Such an example is the `scala-stm-bench7` which is driven by a single thread for ~80% and is multithreaded for the remaining ~20%. Consequently, a trade-off between potential scalability benefits and precise decisions arises. Finally, it is noted that the geometric mean (+0.03x) of all applications confirms that the "stabilization" strategy is beneficial in general.

6 Related Work

The literature on achieving NUMA scalability [1, 4, 6, 8, 18, 19, 29, 30] has identified various limiting factors, including the memory contention over shared resources, congestion on memory controllers and interconnect, and inefficiencies related to page-tables. Numerous works have studied optimizations for the OS scheduling and memory management to increase performance on NUMA systems. However, most of these works have focused on non-managed environments and applications, hence, they study the effect of NUMA architectures in a more direct manner.

Gidra et al. [9] have evaluated the scalability of Parallel Scavenge GC for the JVM and identified memory access imbalance, lack of memory access locality, and contention over shared resources as the main bottlenecks. This work has proposed GC and object placement optimizations to address these issues [9, 10]. Alnowaiser et al.[2] have enhanced GC thread locality, while Patrou et al. [22] have considered thread affinity along with GC to improve the NUMA-awareness of the JVM. Nonetheless, these studies have overlooked the relationship between the properties of the running application and NUMA-related bottlenecks, resulting in a limited understanding of the impact of application properties on memory behavior, scalability, and performance. MacGregor et al. [17] have characterized the memory behavior of GHC and Haskell applications. Papadakis et al. [21] have proposed PerfUtil for low-level NUMA profiling using hardware performance counters, which lacks correlation with high-level application properties.

Thus, the work in this paper aims to complement prior related work by studying the behavior of managed applications on NUMA architectures during mutation time. Additionally, our study is the springboard for the development of a novel optimization mechanism that can increase the performance of Java applications on NUMA systems.

7 Conclusion

In this paper, we studied the behavior of managed applications on NUMA architecture as a means to understand how to scale up their performance. Our study characterized several Dacapo and Renaissance applications during mutation time from both the application and the microarchitectural points of view. The characterization findings resulted in a classification of the running applications into several categories. Then, we capitalized on those findings to implement a novel lightweight and dynamic mechanism in MREs for optimizing the scalability of managed applications on NUMA systems. Our experiments showcased that the proposed mechanism can assess with negligible overhead and in an application-agnostic way, whether an application should scale up on multiple NUMA nodes or it should be deployed on a single node. Finally, the proposed mechanism has yielded in end-to-end performance improvement of up to 3.29x, with a geometric mean of 1.11x, against the vanilla performance of a managed application on a NUMA system.

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). Association for Computing Machinery, New York, NY, USA, 283–300. https://doi.org/10.1145/3373376.3378468

[2] Khaled Alnowaiser and Jeremy Singer. 2015. Topology-Aware Parallelism for NUMA Copying Collectors. In Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519 (Raleigh, NC, USA) (LCPC 2015). Springer-Verlag, Berlin, Heidelberg, 191–205. https://doi.org/10.1007/978-3-319-29778-1_12

[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] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-Box Concurrent Data Structures for NUMA Architectures. In Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (Xi'an, China) (ASPLOS '17). Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

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

[6] 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

[7] 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, 355–372. https://doi.org/10.1145/2509136.2509529

[8] 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). USENIX Association, USA, 231–242.

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

[10] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (Istanbul, Turkey) (ASPLOS '15). Association for Computing Machinery, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[11] James R Goodman and Herbert Hing Jing Hum. 2009. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. Technical Report. University of Auckland.

[12] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control (Portland, Oregon, USA) (AGERE! '14). Association for Computing Machinery, New York, NY, USA, 67–80. https://doi.org/10.1145/2687357.2687368

[13] Reactors IO. 2013. Reactors.IO. http://reactors.io/.

[14] 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

[15] 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

[16] 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

[17] Ruairidh MacGregor, Phil Trinder, and Hans-Wolfgang Loidl. 2021. Improving GHC Haskell NUMA Profiling. In Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing (Virtual, Republic of Korea) (FHPNC 2021). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3471873.3472974

[18] 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

[19] Zoltan Majo and Thomas R. Gross. 2012. Matching Memory Access Patterns and Data Placement for NUMA Systems. In Proceedings of the Tenth International Symposium on Code Generation and Optimization (San Jose, California) (CGO '12). Association for Computing Machinery, New York, NY, USA, 230–241. https://doi.org/10.1145/2259016.2259046

[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, 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). Association for Computing Machinery, New York, NY, USA, 80–88. https://doi.org/10.1145/

3426182.3426189

[22] Maria Patrou, Kenneth B. Kent, Gerhard W. Dueck, Charlie Gracie, and Aleksandar Micic. 2018. NUMA Awareness: Improving Thread and Memory Management. In 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (Prague, Czech Republic). IEEE, New York, NY, USA, 119–123. https://doi.org/10.1109/SEAA.2018.00028

[23] David A. Patterson and John L. Hennessy. 1990. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[24] 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

[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, 139–147.

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

[27] Andrew S Tanenbaum and Albert S Woodhull. 2006. Operating Systems - Design and Implementation - Third Edition. Pearson Education Inc, Upper Saddle River, NJ 07458.

[28] Xinghui Zhao and Nadeem Jamali. 2013. Load Balancing Non-Uniform Parallel Computations. In Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control (Indianapolis, Indiana, USA) (AGERE! 2013). Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2541329.2541337

[29] Xin Zhao, Jin Zhou, Hui Guan, Wei Wang, Xu Liu, and Tongping Liu. 2021. NumaPerf: Predictive NUMA Profiling. In Proceedings of the ACM International Conference on Supercomputing (Virtual Event, USA) (ICS '21). Association for Computing Machinery, New York, NY, USA, 52–62. https://doi.org/10.1145/3447818.3460361

[30] 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 XV). Association for Computing Machinery, New York, NY, USA, 129–142. https://doi.org/10.1145/1736020.1736036

Received 2023-03-03; accepted 2023-04-24