PERFORMANCE ANALYSIS AND OPTIMIZATIONS OF MANAGED APPLICATIONS ON NON-UNIFORM MEMORY ARCHITECTURES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2022

By
Orion Papadakis
School of Computer Science

Contents

Word Count: 32901

List of Tables

List of Figures

Abstract

Scalability of managed applications on Non-Uniform Memory Access (NUMA) architectures has always been a challenging task. Focus has been steered on performance-critical components of the Managed Runtimes such as the Garbage Collectors where NUMA scalability optimizations have been proposed. However, prior to knowing under which circumstances NUMA architecture can be beneficial, the extensive research investment needed for such optimizations would be on quicksand. Moreover, th lack of tooling support for managed runtimes in the context of NUMA puts additional obstacles in the way of analyzing scalability bottlenecks and conclude whether a managed application can benefit from NUMA.

The current thesis studies several memory and scalability aspects of managed applications in the context of NUMA architectures, in order to enable the NUMA scalability of MREs. More specifically, it leverages several Java applications and MaxineVM, a metacircular research VM written in Java. To tackle the lack of tooling support, this thesis proposes a tool-chain composed by the NUMAProfiler, a new Java profiler enriched with NUMA awareness, and by PerfUtil, a microarchitectural profiler with multiplexing support. The effectiveness of the tool-chain is based on the co-utilization of high and low-level profiling tools towards correlating HW metrics with Java application properties. The tool-chain is used to analyze the memory behavior of multiple Java applications picked from two benchmark suites. Moreover, a scalability analysis methodology is presented and applied on those applications in order to characterize them as per their scalability-critical properties. This characterization results in revealing multiple distinct application categories in which typical Java applications can potentially fit.

The research findings that occur from the memory behavior and NUMA scalability studies are formalized into effective NUMA scalability guidelines for improving the performance of a managed application in a NUMA system. The scalability guidelines are amalgamated into a dynamic, application-agnostic, online optimization mechanism

that it is implemented into the runtime layer of MaxineVM. The experimental evaluation of the mechanism showcase that performance ranges from 0.66x up to 3.29x with geometric mean of 1.11x, in comparison to the naive performance of the managed applications on a NUMA system.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Undertaking this PhD has been a truly life-changing experience for me, and it would not have been possible to do without the support and guidance that I received from many people.

I would like to express my sincere gratitude to my PhD supervisor Christos Kotselidis. Throughout those challenging years he was a source of immense knowledge, encouragement, enthusiasm, and motivation. His guidance and patience provided to me the opportunity and space to develop new skills and pursue my own ideas.

I am also thankful to Foivos Zakkak for his thoughtful comments, technical and theoretical support, and recommendations throughout these years. Special thanks to Thanos Stratikopoulos and Nikos Foutris for their valuable help in continuously improving the content and writing quality of my work by reviewing it in detail. In addition, I would like to acknowledge Andreas Andronikakis for our fruitfull collaboration and partnership in numerous aspects all over this journey.

Thanks should also go to the members of the examination commitee, Lucas Cordeiro and Polyvios Pratikakis for thoroughly reviewing my work. I would like to thank Mikel Lujan, the University of Manchester, the EU ACTiCLOUD, E2Data, and ELEGANT projects for supporting this PhD by providing the necessary means.

I would like to express my deepest appreciation to my parents Thalis and Christina, and my brother Iason for their love, interest, and encouragement. Without them, I would never have enjoyed so many opportunities. I would like to thank all the new people and friends I met in Manchester, as well as those friends back in Greece (they know who I am talking about) for all the unforgettable moments throughout these years. These moments have inevitably been an integral part of my PhD journey and you guys are responsible for that. Finally, I would like to specially thank Georgia for her support and unconditional understanding, especially during the most challenging times of the struggle.

List of Abbreviations

ACPI	Advanced Configuration and Power Interface
ALU	Arithmetic Logic Unit
API	Application Programming Interface
BPU	Branch Prediction Unit
CC	Cache Coherency
CPI	Cycles per Instruction
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DSM	Distributed Shared Memory
DVFS	Dynamic Voltage and Frequence Scaling
EA	Escape Analysis
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GC	Garbage Collection
GB	Giga Byte
GHC	Glasgow Haskell Compiler
GPU	Graphics Processing Unit
HW	Hardware
iMC	Integrated Memory Controller
ISA	Instruction Set Architecture
I/O	Input/Output
JIT	Just In Time
JMH	Java Microbenchmark Harness
JNI	Java Native Interface
JVM	Java Virtual Machine
L1	Level 1 Cache
LLC	Last Level Cache

MB	Mega Byte
MMU	Memory Management Unit
MPKI	Misses per Kilo Instructions
MRE	Managed Runtime Environment
NUMA	Non-Uniform Memory Access
OS	Operating System
PC	Program Counter
PCIe	Peripheral Component Interconnect Express
QPI	Intel's Quick Path Interconnect
R/W	Read/Write
RVM	Research Virtual Machine
SMP	Symmetrical Multi-processor
SW	Software
TLB	Translation Lookaside Buffer
TLP	Thread Level Parallelism
UCT	Unbalanced Cobwebbed Tree
UML	Unified Modeling Language
VM	Virtual Machine

Chapter 1

Introduction

Cloud computing and the arising era of the software-as-a-service paradigm push towards servers of larger scale. Modern datacenters are destined to host applications for Big Data, Machine Learning, In-Memory Databases, and more, which tend to be more data-intensive than ever. The ever-increasing amount of data that those applications typically manage have tremendous demands for hardware resources. Therefore, computer architecture is compeled to provide scale-up and scale-out solutions in order to provide sufficient and scalable resources. However, the scale-up techniques of traditional multicore CPUs are not efficient for large-scale systems and applications. The aggregation of multiple cores and one physical memory around a centralized data bus linearly drops the available memory bandwidth per core as the number of cores increases. Therefore, the traditional multicore systems with more than 8-16 cores struggle to efficiently support scalability [PH90]. Non-Uniform Memory Access (NUMA) architecture has been introduced to push forward the aforementioned boundaries. It enables excessive scalability through a more flexible and decentralized hardware layout. It consists of multiple CPUs (nodes) unified into a shared memory system via a high-speed interconnect. However, the existing software is not always capable of exploiting the potential scalability and performance offerings of a NUMA system. As a result, the effort for improving perfromance in the context of NUMA architecture is ever-shifting towards the software-end of the stack.

Progarmming languages, like Java and Python, that rely on Managed Runtime Environments (MREs) offer programmability and safety in development due to the automatic memory management of Garbage Collection (GC), and the hardware and platform-agnostic abstractions. The feature of GC eases the development process because it alleviates the developer from the responsibility of memory management

which can be a harsh and error-prone task. Moreover, the hardware and platform abstractions further simplify the development process because a developer is not required to be aware of the hardware and OS low-level details. For the reasons above, the managed programming languages enhance productivity because they simplify and consequently decrease development time. Hence, they are broadly prefered by the developers. Two out of the top-three most popular programming languages (Python, C, Java) according to the TIOBE[1] and IEEE Spectrum[2] ranking systems, are managed languages. In addition, several programming frameworks for quick and easy development of large-scale applications are built for and in managed languages. Such examples, are the Apache Spark [ZXW+16], and Apache Flink [CKE+15] which are written in Java and Scala, and are leveraged for the development of Java and Scala Big Data processing applications. Managed applications that are developed in these frameworks usually manage massive amounts of memory and therefore, have large memory demands. As a result, the interoperability of MREs with NUMA systems is inevitably a major optimization point regarding the efficient exploitation of large-scale resources by large-scale applications.

This thesis focuses and contributes to this research field by studying the memory behavior of several managed applications, and their NUMA scalability properties. Moreover, it proposes two new profiling tools and a dynamic optimization mechanism for MREs towards improving the performance of a managed application on a NUMA machine. The memory behavior study provides numerous research data and metrics that derive by the co-utilization of the two tools. Those data are analysed and discussed towards drawing reasonable and multi-aspected conclusions that augment the state-of-the-art understanding of the research community for the Dacapo and Renaissance applications. Moreover, this study aims to offer an advanced foundation for future research works, characterisation studies, and optimizations that focus on those managed applications. In addition, the scalability study provides a thorough methodology on which application properties should be assesed under the scope of scalability on a NUMA system. According to the evaluated data this methodology can estimate with confidence the scalability potential of a managed application on a NUMA system, while it is applicable to any managed application. Finally, the optimization mechanism bears the research findings of both studies and stands as a proof-of-concept for novel application-agnostic, low-overhead, and online

[1] https://www.tiobe.com/tiobe-index/
[2] https://spectrum.ieee.org/top-programming-languages/

optimization approaches.

The rest of this chapter is organized as follows: Section 1.1 provides an overview of NUMA architecture along with the traditional challenges that are derived. Section 1.2 introduces the reader to the major challenges for an MRE when run on NUMA architecture. Section 1.3 states the research question, explains the research objectives of the current thesis, and highlights the contributions of the current thesis. Finally, Section 1.4 provides an overview of the thesis structure, while Section 1.5 lists the publications derived from this work.

1.1 Challenges for NUMA architecture

NUMA architecture enhances scalability for systems that aggregate a large amount of CPU cores (more than 8-16). The scalability of this architecture derives from the fact that the computing, and memory resources are distributed instead of centralized. Essentially, NUMA architecture aims to increase the available memory bandwidth per aggregated core. This is achieved by moving away from the monolithic aggregation of resources around a centralized data bus of traditional multicores. Instead, the resources are placed in discrete nodes. Each node is composed by a number of CPU cores and contains its own local memory. The nodes communicate through a low-latency interconnect and provide a globally shared address space across the system (each core can access any memory of any node). This design increases the local memory bandwidth, nevertheless the communication with other nodes becomes more complex. Even though memory of any node is globally shared to any core of the system, higher latency is paid for a remote node memory access. This lack of uniformness requires awareness and special utilization of the hardware by the overlying software stack in order to effectively scale the running application.

The so-called NUMA-awareness, is provided to the software stack in numerous ways. Many Operating Systems, such as Linux and Windows, have been equipped with NUMA-awareness features in order to place data and schedule the processes in a NUMA-friendly manner. For example, the Linux kernel is aware of the hardware topology and aims to place data and threads on the same node wherever possible. In addition, it exposes to the userspace the `libnuma`, a collection of system calls that a programmer can leverage in order to manually control the thread and data placement. Moreover, programming frameworks, such as the OpenMP [DM98] for parallel programming, provide support to the application developer similarly to the

Linux kernel. It is notable that, even though support for NUMA architecture is common across OSs and programming frameworks, expertise is also required by programmers. In order to benefit from the capabilities of a NUMA system, the developer is required to know the principles of this architecture.

1.2 Challenges for MREs in NUMA

1.2.1 Effective NUMA-awareness for the MREs

The managed applications face additional challenges due to the interference of the MRE. A managed application is hosted by an MRE which stands as a layer of abstraction between the developer and the underlying OS and hardware. Typically, the MRE takes over the management of memory and consequently, the developer has no control over where the data is placed. As explained before, the amount of remote memory accesses should be limited in order to avoid the performance penalization of the application. However, if the MRE is unaware of the underlying topology, the allocated memory may be spread across NUMA nodes, while the application threads may manipulate data which is placed on a remote node. These peculiarities bring to the spotlight the objective of enhancing the NUMA-awareness to the MRE itself in order to increase the performance of a managed application on a NUMA system.

Multiple research works have studied the challenge of adding NUMA-awareness in the Java Virtual Machine (JVM), an MRE for the Java programming language. They focus on dynamic runtime features such as the Garbage Collection (GC) and propose numerous NUMA GC optimizations. In addition, NUMA-awareness has been introduced to the OpenJDK HotSpotVM (the current reference implementation of the JVM) by relying on a fragmented - across the NUMA nodes - heap design and allocating the new objects into the heap fragment that is local to the NUMA node that the thread is running (`-XX:+UseNUMA`). Therefore, it is well known that the lack of GC scalability in a NUMA context leads to performance degradation and significant application pause times [GTSS13, GTS+15, AS15, PKD+18].

That being said, NUMA performance for MREs is not only a subject of GC scalability. Running a large set of 30 Dacapo [BGH+06] and Renaissance [PRL+19] applications with MaxineVM [KCR+17] on a NUMA machine (Figure 1.1) reveals that non-GC execution time is significantly penalized as well, by 15% (grey bar), on average, and can degrade performance up to 133% for some applications (i.e.,

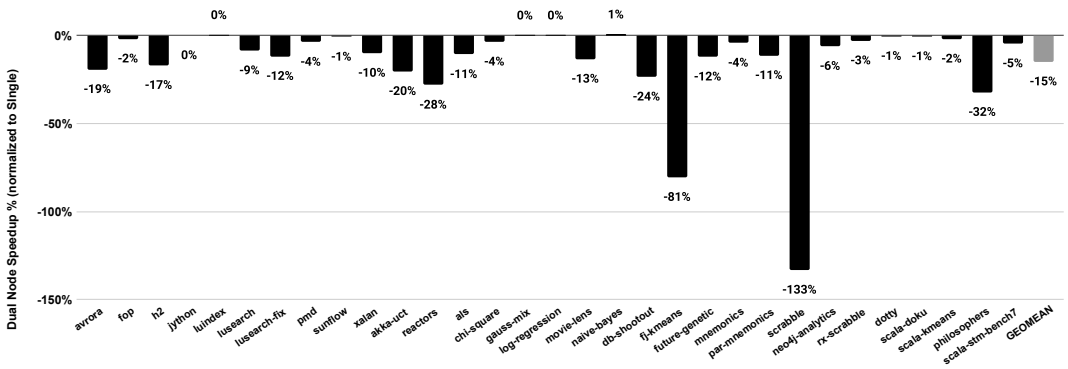


Figure 1.1: NUMA effect on Dacapo & Renaissance benchmarks. Execution time in NUMA normalized non-NUMA.

`scrabble`). Considering the broadly studied Dacapo applications, it is apparent that some are *a priori* unable to benefit from such an architecture due to lack of scalability-prerequisite properties (i.e. parallelism and concurrency) [KMJV12] or even are further penalized due to memory related inefficiencies. Consequently, even with an ideal and linearly scalable GC, it is possible to see no performance gains or, even worse, harm performance. This reality is rather significant considering its potential effect on the Total Cost of Ownership for long-running applications on a large-scale datacenter. Therefore, prior to knowing under which circumstances NUMA architecture can be beneficial, the extensive research investment needed for GC and/or other JVM components optimizations would be on quicksand. As a result, the traditional objective of NUMA optimizations for Managed Runtimes should be further augmented towards understanding under which circumstances NUMA would be beneficial for a Managed application. Hence, a characterization methodology towards deciding whether a managed application is capable to benefit from a NUMA system is needed.

In addition, as Figure 1.1 reveals there is a lot of space for improving overall performance of a managed application on a NUMA system. Optimizing the application performance during mutation can complement a NUMA-aware GC implementation and formalize a jointly optimized JVM for NUMA systems. Consequently, there is a need for effective techniques that optimize the mutation performance of a managed application on a NUMA system.

1.2.2 Tooling support for MREs in NUMA

The process of narrowing down the bottlenecks that bound performance is a prerequisite in order to improve performance. However, the correlation of performance observations with potential bottlenecks is a challenging task. The Hardware Performance Counters that any modern CPU is equipped with, offer various metrics from the microarchitectural layer. However, a microarchitectural-layer profiler lacks correlation between the observed hardware behavior and the application properties, thereby leading to an insufficient or even misleading image[PZFK20]. Moreover, to the best of our knowledge, the Hardware Perfromance Counter support for Java that is provided by some tools such as the Oracle Solaris Studio [Corb], Intel VTune [Cor14a, Don], JMH [Shi], and more, is limited and lack the ability to perform fine-grain profiling.

On the other hand, numerous application-layer tools and Java profilers have been proposed throughout the years. Such examples are JProfiler [Tec], VisualVM [Cor16], AntTracks [LBM15], and more. These profilers focus on memory usage towards mitigating inefficiencies such as memory leaks. Alternatively to those general-purpose profilers are the special-purpose profilers such as FJProf [RRB20], AkkaProf [RCB16], and more that are designed and focus on applications build on specific programming frameworks (i.e., Java Fork/Join [Cor14b], and Akka [Inc09]). A special-purpose profiler can provide specialized metrics that are useful and related only in the context of the target framework. Nevertheless, despite this wide range of tools and profilers for the JVM, to the best of our knowledge none focuses on NUMA architectures. The particularities of a NUMA architecture necessitate an approach that also concerns the NUMA-specific concepts such as remote node memory, etc. Therefore, it is clear that effective tools for the JVM as well as more sophisticated and novel profiling methodologies in the context of NUMA are required.

1.3 Research Objectives & Contributions

Considering the above challenges, this thesis breaks down the overarching question of "how a managed application can take advantage of a NUMA system?" into individual and specific research questions. On that ground, this section outlines these questions and briefly discusses their objectives along with the contributions that derive.

1. **What information is required to comprehend NUMA-behavior in managed**

languages, and which tools, if any, can assist that purpose? As explained earlier, performance analysis is a prerequisite towards improving the performance of i.e., a JVM on a NUMA system. However, lack of tooling support for the MREs in the context of NUMA puts additional obstacles towards research approaches in that field. Moreover, the existing profiling tools typically focus on a single aspect (i.e., application or hardware layer), thereby creating an insufficient profiling image. The current thesis tackles this gap by proposing a novel approach that can effectively profile a managed application in the context of a NUMA system. The proposed approach combines low-level hardware metrics through Hardware Perfromance Counters and high-level application metrics from the runtime layer. A low-level profiling of hardware events, even though insightful, it cannot highlight the root cause of a performance bottleneck observed in hardware (e.g. the LLC Miss Rate increase when an application is run on a NUMA machine). For that reason, this thesis provides a multi-faceted HW/SW profile of an application that analyzes NUMA scalability opportunities and potential optimizations. More specifically, it proposes the co-utilization of the PerffUtil profiler for low-level HW events, and the NUMAProfiler, a Java profiler that also provides insights useful in the context of a NUMA system. This novel approach advances the state-of-the-art by equipping the research community with new sound and efficient profiling techniques. The working principles of this approach are compatible to any MRE even though the current thesis showcases an implementation for MaxineVM and Java applications. The above contributions are briefly presented and showcased in Chapter 4.

2. **Under which circumstances NUMA architecture would be beneficial for a managed application?** To approach this research question, one should consider all application properties related to NUMA scalability. New approaches and methodologies are required due to literature's lack of such studies in the context of NUMA. To that extent, this thesis quantifies the memory behavior and those application properties that allow a managed application to effectively scale on a NUMA system. The main goal is to conclude and formalize a classification methodology that would be applicable on any managed application. Such a methodology would allow to draw a conclusion regarding whether a managed application can benefit from a NUMA system. This research objective is tackled by Chapters 5 and 6. More specifically, Chapter 5 presents a study on the memory behavior of 30 managed applications from the Dacapo and

Renaissance benchmark suites. The memory behavior study advances the state-of-the-art by presenting several new insights of the applications while it reveals a couple of misconceptions regarding the latest literature. In addition, it effectively showcases the benefits of the proposed new profiling approach.

In addition, Chapter 6 presents and apply a methodology on evaluating the potential benefits of a managed application by executing on a NUMA architecture. Thirty Dacapo and Renaissance applications are characterized as per numerous properties critical for NUMA scalability in order to demystify the necessary and sufficient conditions under which NUMA architecture can be beneficial for a managed application. The characterization study concludes into a classification of the applications in categories. Last but not least, the characterization methodology as well as the applied classification of the Dacapo and Renaissance applications contribute towards augmenting the knowledge of the research community, and provide a solid foundation for future research works.

3. **How can an MRE effectively utilize a NUMA system?** The addressing of this research question inevitably derives from the systematic analysis of the aforementioned studies and their results. It essentially amalgamates all the research findings and infrastructure into practical and reasonable techniques for improving the performance of a managed application on a NUMA system. This thesis formalizes reasonable directions for MRE optimizations on a NUMA system. These optimizations are implemented by a dynamic, application-agnostic NUMA-optimization mechanism in the runtime of MaxineVM. The mechanism is briefly described and demonstrated in Chapter 7. It operates during run-time with low overhead (geometric mean = 0.86%) and improves by 11% on average. Therefore, this thesis contributes towards proving the concept of a low-overhead, online optimization mechanism that effectively improves performance. In addition, the same principles are transferable and applicable to any MRE.

1.4 Thesis structure

This thesis is organized as follows: Chapter 2 gives an overview of the key-concepts of MREs, NUMA architecture and Hardware Performance Counters. Chapter 3 further discusses the challenges for MREs in the context of NUMA architectures,

and highlights related research works on that field. Chapter 4 briefly presents the proposed tooling infrastructure. Chapter 5 present a memory characterization study for the Dacapo and Renaissance applications. Chapter 6 presents a NUMA scalability characterization for the Dacapo and Renaissance benchmark suites. Chapter 7 describes and demonstrates a dynamic, and application-agnostic mechanism that improves the performance of a managed application on a NUMA system.

1.5 Publications

This section lists in chronological order, all publications that are related to parts of the current thesis:

- Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, **Orion Papadakis**, Andreas Andronikakis, Iain Apreotesei, Christos Kotselidis. *On the future of research VMs: a hardware/software perspective.*, In Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming (Programming'18) [ZNM+18].

- **Orion Papadakis**, Foivos S. Zakkak, Nikos Foutris, Christos Kotselidis. *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, 2020 (MPLR 2020) [PZFK20].

- **Orion Papadakis**, Andreas Andronikakis, Foivos S. Zakkak, Nikos Foutris, Polychronis Xekalakis, Christos Kotselidis. *Micro-architectural and scalability analysis of managed applications on NUMA systems*. Submited to ACM Transactions on Architecture and Code Optimization (TACO) - [Under Review].

Chapter 2

Background

This chapter aims to provide an overview of the topics related with the thesis. The most relevant areas to this thesis are the Managed Runtime Environments, the NUMA architecture as well as the Hardware Performance Counters. This chapter aims to unfold various sub-topics within the aforementioned areas, such as metacircularity in the JVM, the MESIF cache coherency protocol and more, which are necessary for the readers better understanding. Section 2.1 briefly discuss the major concepts of the Managed Runtimes and explains how a JVM works. It also introduces the reader to MaxineVM and metacircularity. Section 2.2 describes the advent of the Non-Uniform Memory Access (NUMA) architecture and how remote memory affects performance. Section 2.3 introduces the reader to Hardware Performance Counters.

2.1 Managed Runtime Environments

A Managed Runtime Environment (MRE) is a dynamic virtual execution environment that lies in the middle of a Software/Hardware stack, as shown in Figure 2.1. In the context of this thesis, applications written in Java language are deployed consequently the MREs have been used as the prime technology and, more specifically the Java Virtual Machine (JVM) is the utilized MRE of choice. This section presents the fundamentals of MREs through the JVM paradigm. It describes the main components and the workflow of a JVM and discusses the main concepts of MaxineVM [WHVDV+13, KCR+17], a research VM implementation for Java.

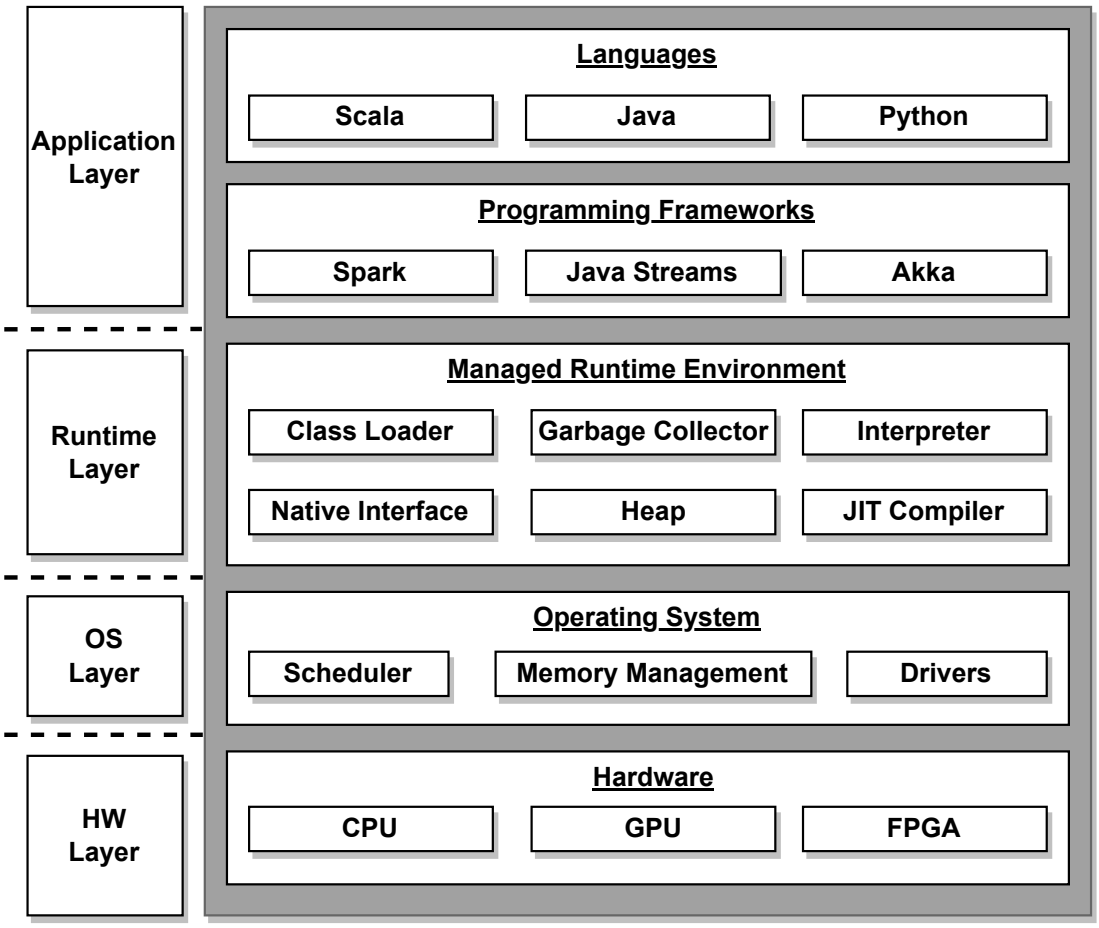


Figure 2.1: An MRE in the context of a typical SW-HW stack.

2.1.1 Java

Programming languages hosted by MREs have emerged as a popular approach for modern parallel applications because they abstract away all the complexity of the underlying hardware allowing developers to stay focused on the logic of their program instead of worrying about hardware specific properties and optimizations. Their hardware-agnostic approach, as well as, other advanced features like automatic memory management/garbage collection (GC) have made them popular and attractive among the software industry. Apache Spark [ZXW+16], Akka [Inc09], Java Fork/Join [Cor14b], Java Streams [Cor14c] and more, are high-level Java frameworks for parallel applications and have brought the MREs and more specifically the JVM to the runtime of choice for modern parallel applications.

Java is a high-level, general-purpose, object-oriented language introduced by Sun Microsystems in May 1995 based on the principle “write once, run anywhere” [Lan02].

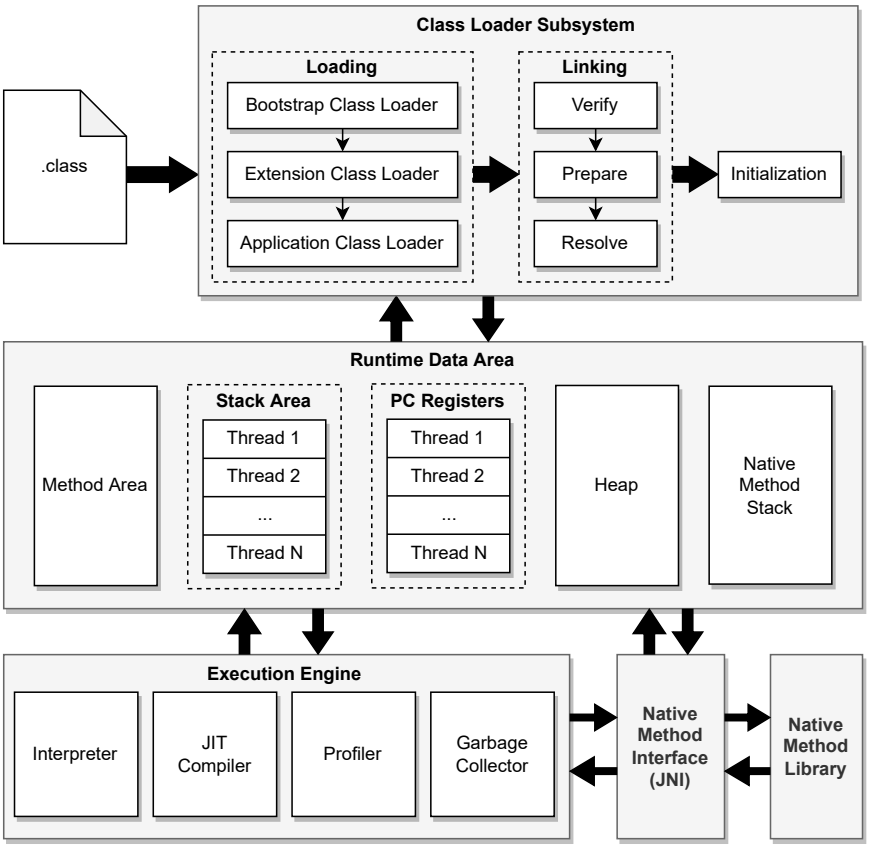


Figure 2.2: Overview of the JVM architecture.

The design of Java abstracts away the underlying Operating System (OS) and Instruction Set Architecture (ISA) from the application itself. A Java program is compiled with the Java Compiler (javac) to a sequence of the so-called Java bytecodes, the intermediate architecture-agnostic instruction-set of the JVM. The bytecode representation is stored in a ".class" file. The bytecodes of a Java program can be executed by any JVM that follows the Java Specification. The JVM implementation is responsible for bridging the gap between the Java program the OS, and the ISA by translating the Java bytecodes to architecture-specific machine code, via "Just In Time" compilation.

2.1.2 Java Virtual Machine (JVM)

The JVM is an MRE that executes programs written in the Java language. In principle, a JVM is "the abstract computer on which all Java programs run" [Ven96]. Essentially, it is a software construction which executes the Java bytecodes and forms a layer between the application and underlying platform (OS & HW). This way the

JVM, stands as a layer of abstraction between the developer and the implementation details of the layers underneath. Any JVM conforms to a set of rules, the so-called specification [Cor06], which defines all the required properties a JVM implementation should bear. This way, Java programs are able to be seamlessly executed on any JVM across different platforms and architectures.

Figure 2.2 shows a high-level overview of the JVM architecture. It is composed of three major components, the Class Loader Subsystem, the Runtime and the Execution Engine.

Class Loader

As a first step, the Class Loader Subsystem loads and reads the ".class" files. This subsystem is responsible for creating a dynamic JVM-internal representation of the loaded classes, methods, and variables in the Runtime Area. Each loaded class is represented as a `java.lang.class` object in the heap. The components of Bootstrap, Extension and Application Class Loader act hierarchically to load the classes, followed by a dynamic linking process. Finally, the class objects are initialized along with their static variables and stored in the heap.

Runtime Area

Upon the completion of the class loading, the Runtime Area contains all the required data for the execution. These data are organized and stored in the Method Area, the Stack Area, the PC Registers, the Heap, and the Native Method Stack (see Figure 2.2). The Method Area is organized per class and holds metadata and the code for methods. The Stack Area holds a stack frame for each thread. Each frame represents the stack of each thread where the temporary variables are stored. The PC Registers component contains a program counter per thread in order to store the address of the currently executed JVM instruction. The Heap is essentially the JVM's memory in which all objects and arrays are allocated. The Native Method Area contains the native method stack that supports the JVM to interoperate with methods written in a native language (e.g., C).

Execution Engine

The Execution Engine is the component that essentially orchestrates the bytecode execution and is typically composed of the Interpreter, the JIT Compiler and the

Garbage Collector. Initially, the Interpreter leads the execution by reading the bytecodes and translating them into machine code ready to be executed. As long as the Interpreter drives the execution, the performance is low due to the continuous and repetitive interpretation (translates the same method more than once); and that this results in the generation of un-optimized code. This low-performance phase is overcome by the optimizing JIT compiler. The JIT compiler monitors (through a profiler) the invocation count of the interpreted methods and compiles to optimized machine code the "hot" methods. It considers a method as "hot" in case the latter is invoked multiple times (beyond the so-called "JIT Compilation Threshold"). This way, the slow interpretation phase is avoided for future invocation(s) of any "hot" method because the optimized JIT compiled machine code is available and ready to be executed. The role of the "JIT Compilation Threshold" is to prevent performance degradation during the JVM start up phase because the JIT compilation consumes high compute and memory resources. Thus, the JIT compilation is not involved in the compilation of "cold" methods (those invoked few times), thereby ensuring a quick startup and that the JVM will be reaching the so-called run-steady phase sooner.

The last key-component of the Execution Engine is the Garbage Collector that implements the "automatic memory management" feature. Typically, the GC is of outmost significance/attractiveness for the MRE and especially the JVM because it alleviates the developer from manual memory management. In its simple form, it is a daemon thread that is responsible for heap space reclamation either upon its fullness (explicit) or upon an application request (implicit). Under both circumstances the Garbage Collector aims to find all the "dead" objects (those are no longer needed) and reclaim the space they occupy. This can be achieved with two alternative strategies: Tracing and Reference Counting. The Tracing strategy considers an object as "live" in case it is reachable via a chain of references starting from certain root objects, so the rest are considered as "dead" and therefore are "collected". The Reference Counting strategy reclaims any object whose reference count (the number of references *to* an object by other objects) is zero.

It should be noted that even though the Interpreter, the JIT Compiler and the Garbage Collector actively participate in Java Bytecodes execution, they are not a subject of the Java specification [Cor06]. A JVM implementation is obliged by the specification only to provide the means for Java Bytecode execution. Therefore, the exact components, their design and implementation are up to the creativity of the JVM developer. As a result, multiple alternative JVM implementations exist and

a large variety of Interpreters, JIT Compilers, and Garbage Collectors are provided by each implementation. The OpenJDK HotSpotVM is currently the *reference implementation* of the JVM. Other notable industrial-strength JVM implementations are the Eclipse OpenJ9 [Ecl], and the GraalVM [WWW+13] which provides the current state-of-the-art Graal JIT compiler. In addition, JikesRVM [AAB+00] and MaxineVM [KCR+17] are notable research VM implementations. JikesRVM offers the so-called MMTk module [BCM04] which contains several state-of-the-art Garbage Collector implementations and a novel modular design to enable fast prototyping of new Garbage Collection algorithms. Nevertheless, JikesRVM has not been actively maintained -by the time of writing this thesis- and lacks of support for modern key-features (such as 64-bit ISA, Java 8, and more) necessary for a state-of-the-art VM.

2.1.3 The Concept of Metacircularity

Graal JIT Compiler, JikesRVM and MaxineVM execute Java programs and also are written in Java. The practice of implementing an execution engine (an interpreter, a compiler or even a whole Runtime Environment) in the same language that it executes applications for is called "metacircularity". The current thesis utilizes the MaxineVM as the language VM of choice for reasons that will be discussed later. The following paragraphs aim to further explain the concept of metacircularity through the example of MaxineVM. Nevertheless, it should be noted that similar practices have been applied to Graal and JikesRVM as well.

As stated above, the source code of MaxineVM is written in Java. Therefore, two steps are required prior to deploying MaxineVM as a VM itself: i) translate MaxineVM Java code to bytecodes, and ii) translate MaxineVM bytecodes to machine code. The latter step implies similar practices to those that will be applied later by MaxineVM (interpretation, JIT compilation, etc.) in order to execute a Java application. Consequently, the major problem of a metacircularity is that the compilation to machine code of the execution engine itself should have been completed at an earlier time ("ahead-of-time").

This "odd" situation is resolved by the concept of bootstraping [RG09]. In the MaxineVM case, the bootstraping is performed by the `BootImageGenerator`, a Java application executed by a pre-existing JVM (the "host" - HotSpotVM in our case) and results to the so-called "Boot Image" [WHVDV+13]. The `BootImageGenerator` re-uses large parts of MaxineVM such as the class loader to create MaxineVM-compatible class representation as well as, the C1X JIT Compiler to translate MaxineVM's

bytecodes to machine code. Throughout this process, several objects are allocated by the utilized subsystems of MaxineVM in the heap of the host JVM. Those objects are carefully collected using the reflection feature of Java and are placed into the synthetic prefabricated MaxineVM heap of the Boot Image. Moreover, the `BootImageGenerator` creates and equips the Boot Image with all the key-components and classes that MaxineVM requires in order to dynamically load classes, compile methods, and execute Bytecodes on its own. After the creation of the Boot Image, the VM is ready for stand-alone execution. Upon launching, the MaxineVM is initiated by a native C program that maps the Boot Image onto the OS virtual memory and passes over the control to the Java code of the Boot Image by an indirect call to a MaxineVM entry point.

2.1.4 MaxineVM

As mentioned above, MaxineVM is a state-of-the-art metacircular **research** VM for Java. As a metacircular software construction, MaxineVM offers increased research productivity (i.e. compared to HotSpot which is written in C/C++) by taking advantage of Java exclusive features and programmability. For example, Garbage Collection simplifies VM development by alleviating explicit and error-prone memory management because the Garbage Collector manages also the MaxineVM objects along with the executed application ones. Moreover, Java's reflection, enables self-introspection capabilities at runtime (i.e. examine object and thread types) which are more than useful for i.e. tooling development.

The design of MaxineVM is modular and flexible to allow easy replacement of existing components or the addition of new. The design of MaxineVM takes advantage of module abstractions, the so-called schemes, to model each component of the VM. This practice isolates low-level implementation-specific details across the schemes while it simplifies their inter-operability; hence, it enhances modularity. The schemes formalize the object layout (`LayoutScheme`), the references (`ReferenceScheme`), the heap allocation and Garbage Collector (`HeapScheme`), the thread synchronization (`MonitorScheme`), the application language environment (i.e. Java) (`RunScheme`) and the compilation policies (`CompilationBroker`). In addition, MaxineVM is equipped with a rich tooling arsenal (i.e. the Maxine Inspector which provides debugging support by correlating low-level entities with high-level Java semantics).

On top of the above, MaxineVM has been selected for this work because it offers

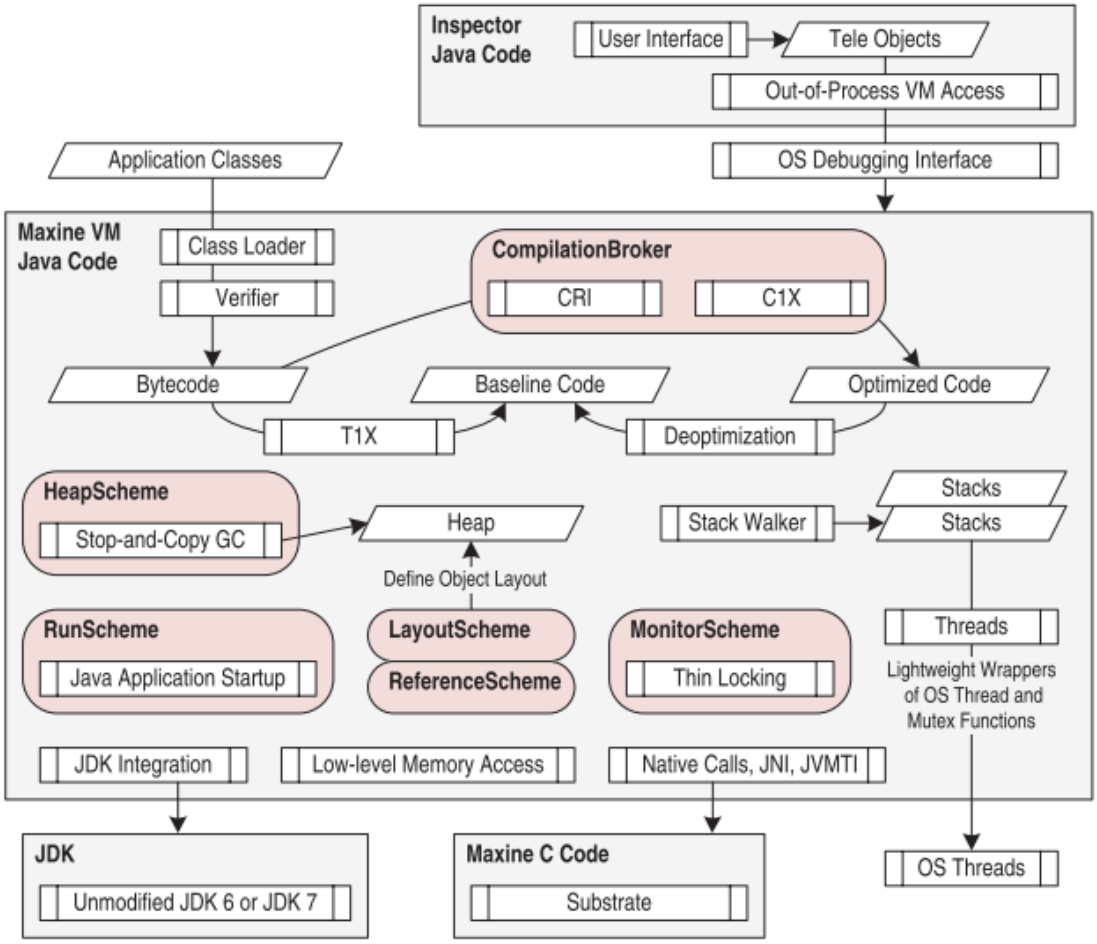


Figure 2.3: Structure of the MaxineVM [WHVDV+13].

a richer environment compared to JikesRVM [RKN+17]. Multi-ISA support (x86-64, ARM v7, ARM v8, RISC-V), two optimizing compilers (Graal and C1X) and T1X interpreter in version 2.9 constitute MaxineVM as a state-of-the-art research VM for Java. Moreover, it is compatible with JDK 8 (while JikesRVM is not) and can run the vast majority of Dacapo-9.12-MR1-bach, Renaissance-0.11 and, SPECjvm2008 benchmarks. MaxineVM achieves 53-57% of the performance of HotSpotVM [RKN+17, WHVDV+13]. Although there is such a performance gap compared to the state-of-the-art, MaxineVM stands as an adequate solution for research in topics not related with peak performance (i.e. compiler optimizations) due to the development productivity it offers.

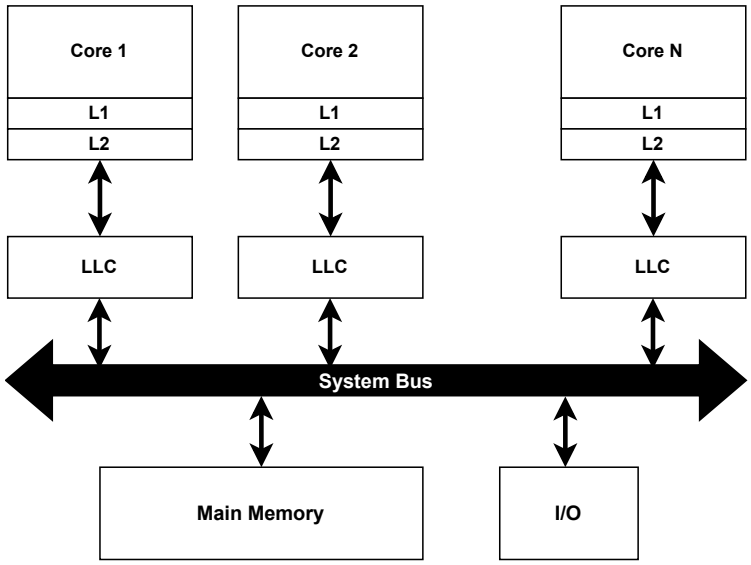


Figure 2.4: Overview of the SMP design.

2.2 NUMA Architecture

Computer engineering problems such as heat dissipation terminated the race for higher CPU clock frequency and, inevitably brought the performance improvements of the traditional single-core CPU to an abrupt end [PH90]. During the early 2000s, the ever-growing number of cores in a single chip arose to mainstream towards improving CPU performance by taking advantage of chip manufacturing technology advancements such as transistor size reduction. Parallelism and scalability were successfully enhanced by the so-called multicore processors or multiprocessors which essentially comprise the "scale up" resources paradigm. As illustrated in Figure 2.4 a typical multiprocessor aggregates more than two cores around a centralized system bus (the so-called front-side bus) with a single, shared main memory providing *symmetrical* memory access latency across cores (Symmetrical Multi-processor - SMP).

SMPs are under the spotlight during the last decades due to the increasing needs of parallel processing power and memory [CCL05, GK06, Ram06, EBSA+11]. However, the SMPs revealed their scalability ceiling soon due to the limited amount of cores a bus can sufficiently serve [CCL05, QYMN09]. As the number of processing units on the bus increases the available bandwidth per core decreases turning the bus-oriented design of an SMP to a bottleneck. In addition, more processing units need longer buses which results to increased latency, thereby turning out the scale up strategy

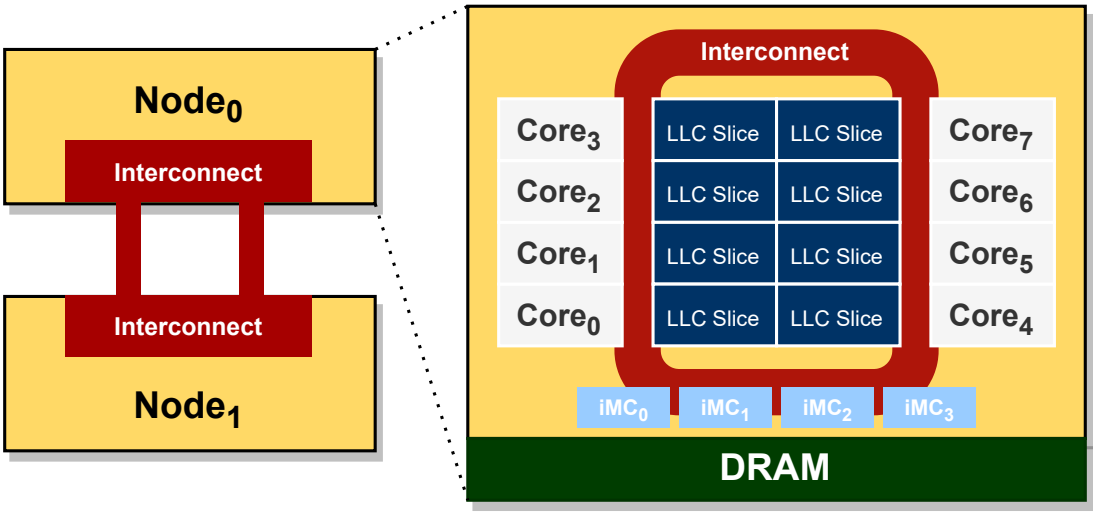


Figure 2.5: Overview of a NUMA system with 2 nodes and 8 CPU cores per node.

to be extremely challenging. To work around these boundaries the so-called many-cores have been introduced and suggested a new, *distributed* and "scale out", resources organization paradigm (Distributed Shared Memory - DSM) [CCL05, PH90].

The Non-Uniform Memory Access (NUMA) architecture is such a design example and enhances scalability over extensive amount of resources by scaling out and organizing them to multiple distinct nodes. The notion of this architecture is to replace the bus-oriented topolgy of an SMP with a more flexible and distributed design in order to enhance parallelism and scalability while avoiding memory's bandwidth saturation at the same time. NUMA reorganizes the entire architecture moving away from the typical SMP design (a monolithic aggregation of multiple processing units) towards a modular building-block design. In a NUMA system, the cores are clustered into nodes and are interconnected through high speed interconnection links (Figure 2.5). 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.

AMD Opteron [OG03] in 2003 and Intel Nehalem [Cor08] in 2007 introduced the first x86 NUMA commodity implementations bearing two significant changes regarding the memory controller and the intercommunication between CPUs. Both microarchitectures introduced multiple memory controllers per node (firstly by AMD Opteron). Each CPU is equipped with one or multiple dedicated memory controller(s) as well as Last Level Caches (LLC) and Memory Banks; the *"Uncore"* (as it was introduced by Nehalem). This design transforms the CPU to an individual building

block for the entire system. The cores access memory with uniform latency into the premises of a CPU. However, a low-latency and high-bandwidth point-to-point connection replaces the restrictive system-bus and can link multiple CPUs to create systems of larger scale compared to an SMP. Intel and AMD introduced their own interconnect implementation, the Intel QuickPath (QPI) [Cor20] in Nehalem and the AMD HyperTransport [AMD19] in Opteron accordingly.

As Figure 2.5 abstractly illustrates, each node encompasses a discrete multicore CPU with local DRAM and a memory controller which communicates with the other nodes through a high-speed point-to-point interconnect (e.g. Intel's QuickPath (QPI), AMD's HyperTransport, etc.). Even though the LLC and memory are distributed across the nodes, the modern NUMA implementations aim to provide a unified and globally shared memory address space [CCL05]. Those implementations are called cache coherent NUMA (ccNUMA) because they deploy cache coherency mechanisms to provide shared memory. The following sections present the major cache coherency mechanisms (Section 2.2.1), and the components a NUMA node is composed of (Section 2.2.2)

2.2.1 Cache Coherency

As stated earlier, modern NUMA implementations aim to provide a shared memory environment. ccNUMA systems are typically easier to program since they do not require an alternative programming model. Cache coherency is a key requirement for *distributed* designs, such as NUMA, because a running program will normally place copies of the same data in multiple caches across the system [PH90]. Lack of coherency between the read/written cached data leads to outdated data accessing/processing (the so-called coherency problem) and subsequently, lack of memory consistency [PP84]. Consequently, a NUMA system inevitably needs a mechanism to maintain coherency across the caches of multiple nodes, the so-called cache coherency mechanism. The high complexity and sensitivity of a cache coherency mechanism has resulted in a very careful and strict formulation of their algorithmic behavior, usually through a protocol [PH90, PP84, GH09]. The protocols define the steps required for a cache line to transit from a sharing state to another (i.e. from Modified to Invalidated) and can be represented by a Finite State Machine. The "directory-based" and "snooping" are the two most common protocol classes for a cache coherency mechanism [PH90]. Both protocols aim to track the status of any block of data that is shared across multiple cores. However, they use different

techniques. A brief description of each protocol class follows:

- *Snooping Protocols*:

  Small chunks of information (snoops) are broadcasted from a node upon the occurrence of any request. The snoops circulate in the interconnect bus to reach all nodes consuming valuable bandwidth and increase traffic congestion undermining scalability ($\sim$ beyond 64 nodes). A transaction needs only two steps (request/response) to be completed, thereby turning snooping protocols to a fast and easy-to-implement solution. The simplest snooping protocol is the MSI and is based on three individual cache line status states: Modified, Shared, and Invalid [PH90]. Nevertheless, many variants of the MSI protocol exist and introduce additional states to optimize its behavior for better performance. Such examples are the popular MESI [PP84] (adds the Exclusive state to reduce snoops for local cache reads), the MESIF [GH09] (extends MESI with the Forward state to enable multiple copies of shared read cache lines) by Intel and, the MOESI [AMD20] (extends MESI with the Owned state to reduce memory write backs for already Shared cache lines) protocol by AMD.

- *Directory-based Protocols*:

  These protocols depend on multiple *distributed* directories (for DSMs) that store information regarding the sharing status and the physical location of every cache line. Upon a node request, the directory forwards it to a specific node leading to point-to-point messages on a "need-to-know" basis instead of broadcasting; thus directory-based protocols are more sufficient for large-scale NUMA machines (i.e. NUMAConnect [Rus]). Directory-based protocols suffer from longer latencies than snooping because a transaction is composed of three steps (request-forward-respond).

Even though snooping protocols tend to bound scalability beyond a number of NUMA nodes, they are highly efficient for systems with few nodes due to low complexity and the fact that they perform transactions in two steps. Moreover, the optimizations that significantly reduce the broadcasted snoops and consequently diminish the interconnect traffic have made snooping protocols more attractive. The following paragraph discusses the key features of the MESIF protocol as it is the one deployed by the NUMA machine used in this thesis.

MESIF Protocol

As stated earlier, this protocol extends the MESI protocol, and it was firstly introduced by Intel to implement QPI [GH09]. It contains the following cache line states:

- The **Modified** state denotes that the data is cached only on one cache and its value has already been modified (dirty). Note that main memory should be firstly updated (write-back) upon read request for the outdated main memory data. Then the Modified cache line transits to Shared state.
- The **Exclusive** state expresses that the cached data match the main memory value (clean). It may transit to Modified upon a write request or to Shared upon a read request.
- The **Shared** state indicates that the cache line is clean but present in other caches.
- The **Invalid** state flags that the cache line is currently outdated (unused).

The MESIF protocol aims to take advantage of the fact that data can always be fetched faster from cache - even from remote node- than memory. For that reason it expands the MESI protocol by introducing the **Forwarding** state which essentially is a "special" Shared state. The MESI protocol allows a Shared cache line to be present in multiple nodes. In that occasion, the MESIF protocol nominates one of those nodes to be responsible for responding with the data upon a request instead of memory [GH09]. The MESIF protocol achieves that by setting this cache line of the nominated node to the Forwarding state.

2.2.2 NUMA Node Organization

Figure 2.6 illustrates the components of an Intel Sandy Bridge[1] NUMA node that are related to processing, memory and cache coherency. It is composed of cores, LLC slices (B-Box), C-Boxes, Memory Controllers (iMC), the Home Agent, the P-Box and the QPI interconnect [Cor12b]. In addition, it contains a power controller (PCU), an integrated IO (IIO), PCIe x8, and PCIe x16 [Cor12b] which are not further discussed since they do not concern the current work. The C-Box module implements the cache coherence infrastructure. The Home Agent which bridges the iMCs with the interconnect. The P-Box contains the QPI interfaces responsible for

[1] The microarchitecture used in this work.

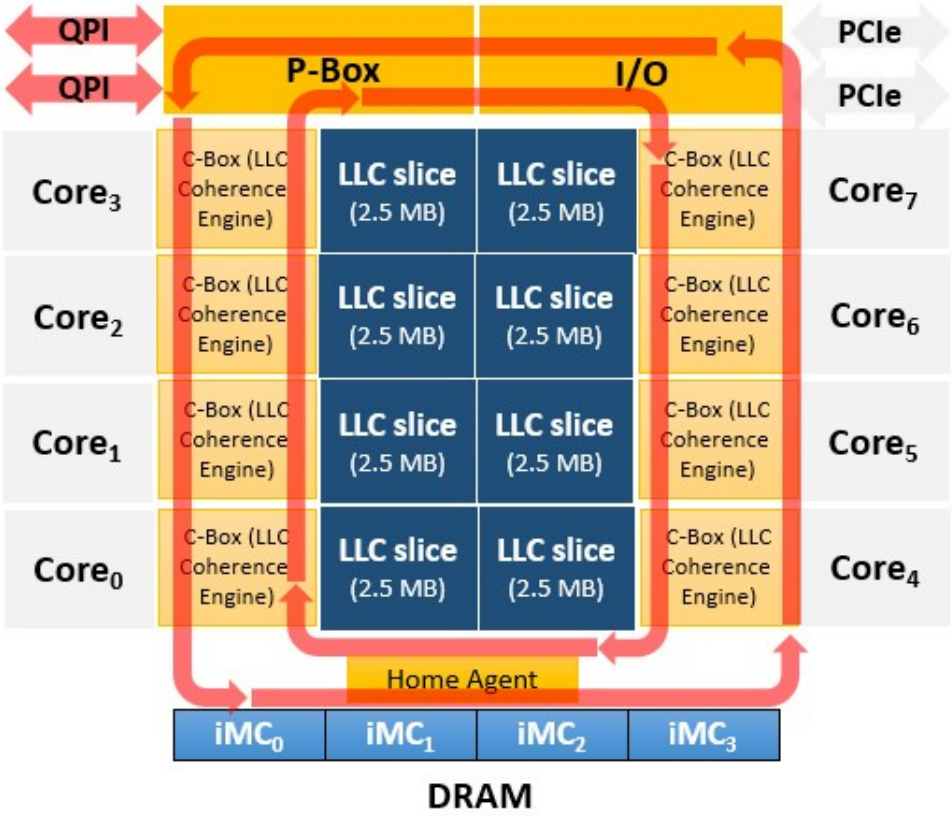


Figure 2.6: A typical NUMA node architecture (Inspired by [Cor12b]).

the communication with the other NUMA nodes. The incorporation of above modules creates a shared pool of individual resources aggregated around a low-latency high-bandwidth two-way interconnect (QPI for Intel) across all NUMA nodes. The several sub-modules of each NUMA node are not only shared into the scope of a single node but are also "visible" and shared across multiple nodes of the system. The above modules - except the cores - comprise the so-called Uncore [Cor12b].

2.3 Hardware Performance Counters

Collecting information and correlating them with the performance of an application has always been a challenging task with numerous alternative approaches (such as platform simulation, instrumentation, application-layer profiling etc.). The introduced overhead as well as the potential distortion of the application's behavior are traditionally the main constraints for any profiling approach. Considering the above, since mid 90s the modern CPUs have been equipped with some special-purpose registers able to

count a wide-range of hardware events. The so-called Hardware Performance Counters (HPCs) (or Performance Monitor Units (PMUs) according to Intel terminology), have been introduced in the Intel Pentium microarchitecture [Cor11a] and have been established throughout the years as an attractive profiling mean. Nevertheless, they can be found in CPUs of other vendors as well, such as AMD [AMD20]. The Hardware Performance Counters essentially are some special-purpose registers in the CPU die that offer low-overhead readings of hardware-related event counts (i.e. CPU cycles, instructions, etc.) [DS11, ZLR16]. Moreover, they minimize the distortion of hardware behavior because they do not interfere or block the pipeline. In addition, can be handled with low overhead [Wea13]; hence they stand as an effective and useful profiling mean [DS11, DEKB16, ZLR16].

The Hardware Performance Counters are programmable, hence they can monitor a wide range of performance-critical micro-architectural events, as defined from the silicon manufacturer. The most common events are related to typical CPU components such as the cache and TLB (e.g., accesses and misses at all levels), the cores (e.g., retired instructions, clock cycles, etc.), the Branch Prediction Unit (e.g., branch instructions, branch misses) and more. However, the support for the available hardware events has been increased throughout the years, as the shipped processors evolve [DS11]. Modern complex multiprocessors are usually composed of several subcomponents (such as the Uncore) which are equipped with their own set of dedicated Hardware Performance Counters [Wea13]; thus providing an ever-increasing and broad set of available "events" for one to count. The NUMA systems follow this trend and offer Hardware Performance Counters for their special components such as the interconnect, etc. [Cor12b]. The utilization of Hardware Performance Counters is achieved by ISA-specific hardware instructions (e.g. the RDPMC for x86). They are deployed by numerous profiling tools and frameworks [DEKB16] such as PAPI [BDG+00], Intel VTune [Cor14a], and Linux Perf [EGMdB15a, Wea13].

Linux Perf is utilized as the framework of choice by this thesis for deploying the Hardware Performance Counters because it is included to the Linux kernel; hence no additional library and installation is required. Linux Perf is a framework/tool of the Linux kernel. Perf is present in Linux kernel since version 2.6.31 in 2009 [Edg09]. Perf handles the low-level ISA-specific instructions for the user-defined Hardware Performance Counters under the hood. On its simplest form Perf can profile any executable with the `perf stat` command, as shown in Listing 2.1.

However, `perf stat` is a "coarse-grain" approach since it profiles the program and

Listing 2.1: Perf stat example for the `ls` program

```
user@machine:~$ perf stat -e cycles,instructions,branches ls
Directory1  Directory2  Directory3  file.txt code.c

 Performance counter stats for 'ls':

 2,248,079    cycles
 1,417,899    instructions      #      0.63 insn per cycle
   278,393    branches
```

monitors the counters of choice for the whole end-to-end execution. For more fine-grain profiling, Perf exposes an API to the user-space through the `perf_event_open` system call. The events are handled (enable, disable, reset) with the `ioctl` (or `prctl`) system calls and the event values are read with the `read` system call. The utilization of that API can offer "fine-grain" monitoring since it delegates the control for when to start/stop profiling to the programming layer. As a result, a (native language) developer can bind the code section of interest with Perf monitor capabilities in a straightforward manner and obtain the desired profile. Such an approach is a necessity especially in the context of MREs to mitigate as possible the effect of the MRE layer and isolate the application behavior. In addition, enabling a JVM to interface with Perf API would provide multiple research opportunities. More specifically, the Hardware Performance Counters utilization from the JVM runtime layer would allow "fine-grain" profiling of the running application with low overhead, and even drive dynamic optimizations. However, the Perf API utilization for a managed application is not as straightforward as in the case of a native application because an additional mechanism that interfaces with the native Perf API is required. Unfortunately, such a feature is rare across the existing JVM implementations (to the best of our knowledge is supported only by JikesRVM) and apparently it is not enclosed in the JVM specification. This fact combined with the potential opportunities that such a feature could enable motivated us to implement the utilization of Hardware Performance Counter for MaxineVM (see Chapter 4).

Chapter 3

Related Work

3.1 NUMA-awareness in the Linux kernel

While the lack of effective and efficient NUMA-awareness in Linux has been acknowledged as an important problem throughout the years, little progress has been made so far. The complexity and side effects of such an approach which inevitably involves multiple critical OS components (kernel, scheduler, virtual memory etc.) prevents many ideas from being adopted. However, Linux kernel is equipped with an automatic page migration mechanism (NUMA balancing), four memory allocation policies (default, bind, interleave, and preferred), a user-level library (libnuma) and a bunch of thread and memory affinity system-calls [CSTL19].

NUMA balancing [Cor12a], [Cor13] is an automatic page migration mechanism in the recent versions of the Linux kernel (after 3.8). This technique lazily reveals the pages that need to be migrated by exploiting page-faults. After the initial allocation, the scheduler invalidates the MMU entries of the process, in order to trap the page with an "artificial" page-fault at its *next-touch*. If a process touches the "marked" page, the kernel becomes aware that the page should be migrated at process's home node, through the handling of the page-fault. This lazy mechanism restrains unnecessary page migrations (e.g., if a page is not touched again, or touched by a process on the same node) while it also avoids space consuming data structures and complex algorithms that are needed in order to track each memory page during execution. However, the overhead of migrating a 4 kB memory page is significantly higher than a remote node access [YAR09]. 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. Moreover, by targeting only to reduced remote memory accesses might lead to inefficiencies of higher importance such as increased interconnect and memory controller congestion [DFF+13].

3.2 Traditional NUMA scalability issues

Numerous works over the years have highlighted the major bottleneck factors that bound NUMA scalability as well as, the research and engineering effort need to be invested towards mitigating drawbacks and exploiting the benefits of NUMA [ZBF10, Kam11, MG11, MG12, DFF+13, GLD+14, CSBA17, APB+20, ZZG+21]. Cache and memory locality, contention over shared resources, congestion over the memory controllers and the interconnect as well as inefficiencies related to the page-tables have been identified as the major NUMA scalability boundaries, and even performance degradation factors. For example, Zhao et al. [ZZG+21] propose a new NUMA profiler capable in detecting NUMA-related performance issues such as memory controller and interconnect congestion as well as cross-node migration. On the other hand, scheduling and memory management mechanisms of the OS are the traditional candidates for NUMA optimizations due to their straightforward capabilities in adopting awareness of the underlying hardware topology. More specifically, Zuravlev et al. [ZBF10] optimize scheduling to mitigate shared resoures contention based on a thread classification scheme. Kamali [Kam11] implements a scheduling algorithm that clusters and co-locates the threads that share data by utilizing some special Hardware Performance Counters that are related to the snooping protocol. This technique, though novel, it is efficient only for applications with two threads because the utilized Hardware Performance Counters can only reflect the effect of data sharing and cannot distinguish the engaged threads. Majo et al. [MG11] take under consideration both memory locality and process scheduling towards NUMA-friendly scheduling. Dashiti et al. [DFF+13] argue that memory controller congestion is rather critical for NUMA performance and refactor Linux Scheduler accordingly. Calciu et al. [CSBA17] relax contention over shared data structures by implementing Node Replication (NR), an application-layer API that can automatically transform a sequential data structure into a concurrent NUMA-aware one in exchange for memory consumption multiple of the number of NUMA nodes. Gaud et al. [GLD+14] claim that modern OS features such as large pages limit performance gains even under NUMA-locality and

balance maintenance scheduling, and they extend [DFF+13] towards that objective. Finally, Achermann et al. [APB+20] implement NUMA-aware page-table placement to mitigate excessive remote node accesses as a result of TLB misses.

The above studies converge on the fact that they study non-managed applications, hence they are able to study the effect of NUMA architecture on HW/SW interoperability in a more direct and transparent manner.

3.3 NUMA scalability and optimizations in the context of MREs

The analysis of performance-related issues for managed applications as well as their correlation with the underlying hardware architecture has posed several challenges. The MRE itself, as an additional layer, is fused within the application's observed memory behavior. In addition, MRE features, such as the Garbage Collection (GC), interfere on the memory patterns of an application. Therefore, a "divide and conquer" strategy towards analyzing and optimizing managed applications in the context of NUMA is a necessity. As such, the key analysis and optimization points are: a) memory behavior during mutation, which is directly related to application properties and characteristics; and b) the GC algorithm/strategy.

The current state-of-the-art for the referenve JVM implementation (HotSpotVM) is the `-XX:+UseNUMA` option. This option enhances the NUMA-awareness of the Parallel Scavenge (default until Java 8) and G1 collectors (default from Java 9 onwards[1]) [Cor11b, Kim20]. The working principle of this option is to split the heap into regions (G1 organizes the heap in regions by default) and evenly spread them across the NUMA nodes. This design aims to enhance scalability by taking advantage of the object placement in the local node of the thread.

In addition, multiple research works have analyzed performance issues of managed applications in the context of NUMA, as well as proposing NUMA optimizations [GTSS13, GTS+15, AS15, PKD+18]. Gidra et al. [GTSS13] assess the scalability of Parallel Scavenge GC and highlight memory access imbalance, lack of memory access locality and contention over shared resources (locks) as the major bottlenecks. Moreover, they propose numerous NUMA GC and object placement optimizations in [GTSS13, GTS+15]. Alnowaiser et al. [AS15] proposes techniques to

[1] The `-XX:+UseNUMA` option was enabled for G1 from Java 14

improve the locality of GC threads. Patrou et al. [PKD+18] take under consideration also thread affinity along with GC.

The above studies converge on the perception that the GC is the major NUMA bottleneck source, hence they focus on it. They lack in corelating application properties with NUMA bottlenecks and therefore identifying under which circumstances a NUMA system can be beneficial for a managed application. This, inevitably, results in limited understanding/discussion for applications that the proposed sophisticated GC optimizations show negligible or even negative impact. Such an observation further motivates the research gap that this thesis covers in Chapter 6. The properties and characteristics which are necessary and sufficient for a managed application towards effectively exploiting a NUMA system still remain open research fields in which the focus of this thesis is steered.

3.4 Profiling Tools & Characterization Studies

Many research efforts [KMJV12, DBSEE13, RCB16, LBM15, LBMW17, RRB20, MTL21] have aimed to analyze the performance-critical properties of managed applications. Some of them have proposed new profiling tools for the JVM, such as AntTracks [LBM15], AkkaProf [RCB16], and FJProf [RRB20] as well as novel profiling techniques, such as bytecode instrumentation [KMJV12], runtime-driven JVM instrumentation [KMJV12], and BottleGraphs [DBSEE13]. Kalibera et al. [KMJV12] exploited bytecode instrumentation and runtime-driven JVM instrumentation to study a wide set of concurrency metrics for Dacapo benchmark suite. DuBois et al. [DBSEE13] leveraged BottleGraphs and studied the exhibited parallelism of Dacapo benchmarks. Lengauer et al. [LBMW17] utilized AntTracks to study the memory behavior of Dacapo, Dacapo Scala and SPECjvm2008 benchmark suites. AkkaProf and FJProf are two special-purpose profilers [RCB16, RRB20] utilized for providing effective profiling metrics for Akka and Fork/Join-based Java applications. Unlike the aforementioned studies and tools which have not targeted NUMA architectures, MacGregor et al. [MTL21] proposed NUMA profiling techniques and characterized memory behavior. However, they focused on the Glasgow Haskell Compiler (GHC) and on Haskell applications. In addition, it is notable the limited choices in the available tooling infrastructure regarding the utilization of Hardware Performance Counters for Java applications. Moreover, the few and rare implementations either lack the ability to perform fine-grain profiling, such as

the Oracle Solaris Studio [Corb], Intel VTune [Cor14a, Don], JMH [Shi] or even are not actively maintained, such as the JRockit [Cora], and the JikesRVM [AAB+00] which both lack of support beyond Java 6. The tools proposed by this thesis in the Chapter 4 aim to tackle the scarcity of tooling support regarding the NUMA architectures and Hardware Performance Counters for the managed applications.

Chapter 4

Memory Profiling of Managed Workloads: A HW/SW Perspective

4.1 Introduction

Memory profiling for managed applications is a challenging task due to the "noise" introduced by the MRE itself. This additional layer inevitably increases the imprecision of the already coarse-grained black-box profiling techniques (e.g., `perf stat` [EGMdB15a]). Moreover, other profiling techniques are rather inefficient in the context of managed applications. For example, wrapping the application code section of interest with Perf system calls, would affect the normal behavior of the application due to the required JNI calls [DJL09], and would introduce considerable overhead [Wea13] especially under frequent utilization. In addition, requires modifications of the application code, which is a harsh task, especially in the context of modern managed application that are built in complex frameworks. Therefore, traditional and easy-to-apply profiling techniques from the native world, lack of accuracy and applicability in the context of managed applications. Moreover, the increased complexity that comes with MREs requires multiple layers of a system stack to be monitored concurrently [PZFK20], thereby demanding more sophisticated methodologies that would go beyond the state-of-the-art. However, associating the underlying hardware behavior with the characteristics of managed software and vice versa is a non-trivial task. Nevertheless, this is a necessity in the context of NUMA architecture [PZFK20].

This thesis proposes a novel approach that aims to effectively correlate the application properties with hardware behavior, in order to achieve a better

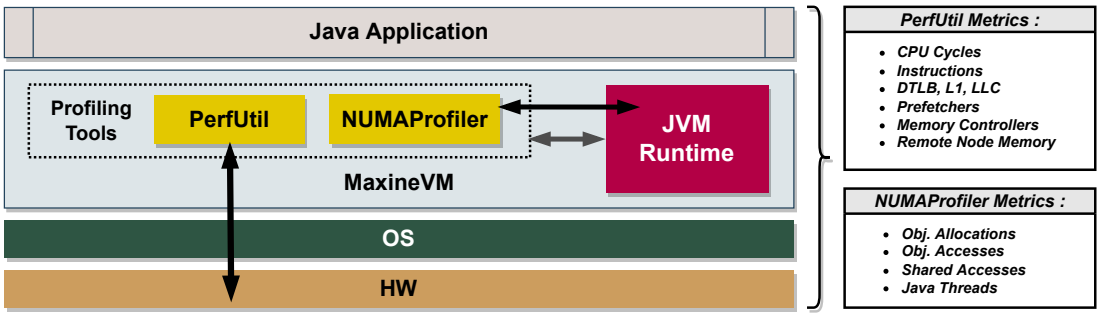


Figure 4.1: An abstract figure of the proposed research platform.

understanding of the behavior of managed applications when deployed on a NUMA system. For that reason, this chapter presents two new tools, that compose the research platform that is proposed and utilized by this thesis. More specifically, the two tools are:

1. the *PerfUtil*, a low-level microarchitectural profiler, and
2. the *NUMAProfiler*, a high-level application-layer profiler.

In particular, PerfUtil monitors the hardware performance counters, while NUMAProfiler probes the runtime layer of MaxineVM and monitors object-related metrics. Figure 4.1 abstractly illustrates the two tools within MaxineVM and provides an overview of the metrics that each one can provide.

The rest of this chapter describes in detail the two profiling tools of the proposed research platform and discusses the challenges faced and the key design choices taken. More specifically, Section 4.2 describes the PerfUtil and Section 4.3 presents the NUMAProfiler.

4.2 PerfUtil

The Linux kernel exposes the utilization of the Hardware Performance Counters to the user-space via the perf tool. However, the effective profiling of a managed application with as higher precision as possible, is challenging; especially as an external observer (i.e., monitor the whole process with `perf stat`). As explaned in Section 4.1, such an approach is considered as coarse-grained, and therefore, ineffective. Moreover, especially in the context of managed runtimes, it leads to the following inefficiency. An external observer that monitors the whole process is unable to focus on the behavior of the application itself because the profiling results inevitably incorporate the MRE

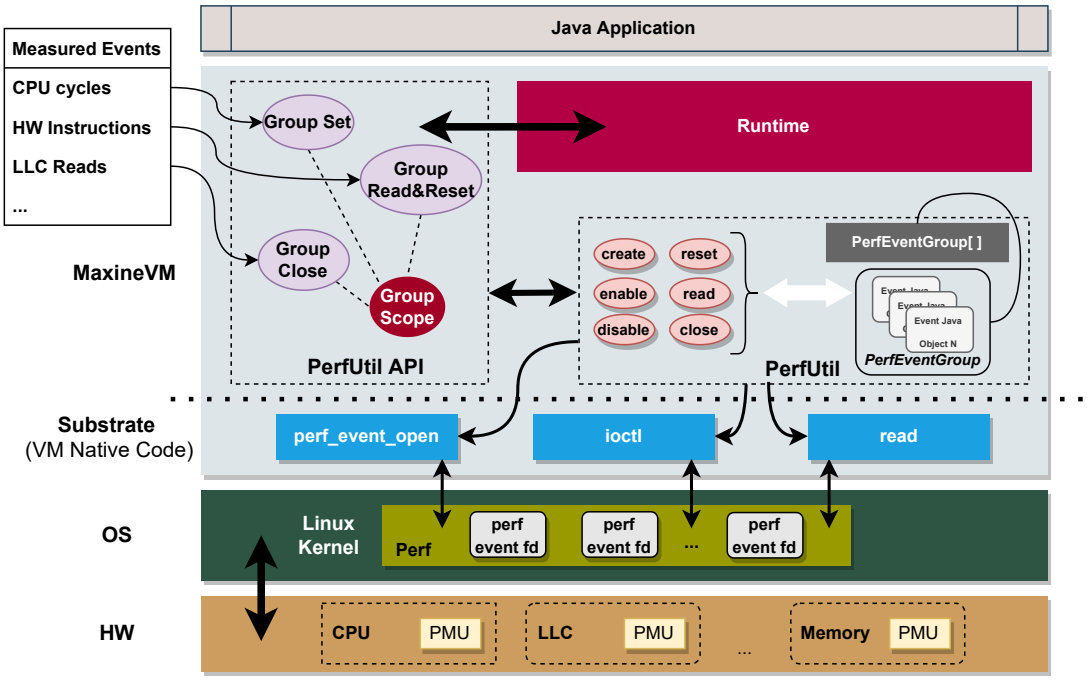


Figure 4.2: PerfUtil Overview.

behavior *and* the application behavior. Therefore, a profiling technique capable of isolating and monitoring *only* the application behavior is required in order to increase the precision and the effectiveness of the profiling results. To address that challenging task, this thesis proposes PerfUtil, a new MaxineVM component to equip the VM itself with fine-grained utilization of the Hardware Performance Counters.

PerfUtil interfaces the Perf API of the Linux kernel and passes over the control to the Java code of the VM. Perf is implemented into the Linux kernel space, and therefore, it is not directly accessible from the user-space. It exposes a higher-level native API to deploy the hardware events from the user-space, and consequently it abstracts away the low-level utilization of the Hardware Performance Counters (see Section 2.3). PerfUtil leverages the Perf API from the user-space, and finally, it bridges the utilization of Perf events within the Java world.

PerfUtil is illustrated in Figure 4.2 where the major components and the interoperability with other layers of the stack (Runtime, OS, HW) are highlighted. As can be observed it can be broken down into two parts, the *PerfUtil module* and the *PerfUtil API*. The PerfUtil module implements all the core functionalilty of the tool (see Section 4.2.1), while the PerfUtil API supports the interoperation of the PerfUtil module with the VM Runtime (see Section 4.2.2).

Listing 4.1: The utilized system calls by the MaxineVM substrate to interoperate with perf

```
// creates a file descriptor that corresponds to one perf event.
int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu
    , int group_fd, unsigned long flags);

// manipulates a perf event through its file descriptor.
int ioctl(int d, int request, ...);

// reads a perf event's value through its file descriptor.
ssize_t read(int fd, void *buf, size_t count);
```

4.2.1 PerfUtil Module

The PerfUtil module contains all the neccessary classes, methods, and data structures to support all the core functionalities for manipulating a Perf event and a group (create, enable, disable, reset, read, close). It essentially models each Perf event as an individual Java object (`PerfEvent`). An event object holds a `MAXINE_PERF_EVENT_ID` value which is a PerfUtil-internal ID that correlates the `PerfEvent` object with the low-level Perf event configurations. In addition, the PerfUtil module implements all the necessary native functionality to interoperate with the OS via the MaxineVM substrate It essentially extends the native substrate of MaxineVM to include all the system calls needed (`perf_event_open`, `ioctl`, `read`) for a `PerfEvent` manipulation. Listing 4.1 provides the signature of the utlized Linux system calls.

Group-oriented design

PerfUtil also supports group configurations. The notion of grouping implies that a set of semantically related events are simultaneously counted during the same period of time so that their values are comparable. Such a technique is necessary for more complex Perf configurations in order to avoid incosistencies in the profiling results (i.e., more LLC misses than LLC accesses). Perf checks under the hood whether all the group events can actually fit in the available physical Hardware Performance Counters; if yes, that group is successfully scheduled, if not, the measurement is aborted.

The design of PerfUtil is aligned with the above event group management strategy. A collection of event objects shapes a group object `PerfEventGroup`. All `PerfEvent` objects that are related to a `PerfEventGroup` are stored in an array

Listing 4.2: PerfUtil API

```
void initialize();

/* Group Set methods */
void perfGroupSetSpecificThreadSpecificCore(int threadId, int core)
void perfGroupSetSpecificThreadAnyCore(int threadId)
void perfGroupSetAnyThreadSpecificCore(int core)
void perfGroupSetAnyThreadAllCores()

/* Group Read & Reset methods */
void perfGroupRnRSpecificThreadSpecificCore(int threadId, int core)
void perfGroupRnRSpecificThreadAnyCore(int threadId)
void perfGroupRnRAnyThreadSpecificCore(int core)
void perfGroupRnRAnyThreadAllCores()

public static void explicitPerfGroupReadAndReset()

public static void perfGroupClose(int threadId, int core)
```

into the `PerfEventGroup` object. The PerfUtil module contains a data structure (`perfEventGroups`), which is a simplified custom hash table, that stores the groups. Each group is uniquely indexed in the hash table. The index of each group in the array is computed by a hash method therefore, each `PerfEventGroup`, and subsequently each `PerfEvent`, can be retrieved any time with O(1) complexity. This design choice has been taken due to the minimal overhead it requires in indexing a group. Keeping the overhead and distortion of the execution as low as possible is of outmost importance and a conscious choice with the aim of re-using PerfUtil for online dynamic optimizations.

4.2.2 PerfUtil API

The functionality of the PerfUtil module is exposed to the VM by the PerfUtil API. The API lies in the runtime layer (MaxineVM code) and contains a collection of Java methods to make the functionalities of PerfUtil module exploitable from a high level programming language, such as Java. The API provides three categories of methods:

- the `PerfGroupSet` for initializing the low level Perf events and the corresponding objects for a new `PerfEventGroup`,
- the `PerfGroupReadAndReset` for reading and reseting the value of an already active `PerfEventGroup` and,

- the `PerfGroupClose` for closing all the events of a `PerfEventGroup`.

These methods use the PerfUtil module methods for group manipulation as building blocks. For example a `PerfGroupSet` API method consists of the `create()`, `reset()` and `enable()` methods of a `PerfEventGroup` which are implemented in the PerfUtil module. Note that, the event value is reset right after its creation to avoid any stale values.

As can be seen in Listing 4.2, multiple versions of each API method category exist. Each version essentially reflects the alternative available "scopes" that a `PerfEvent` can have. A `PerfEvent` can be counted per thread, per core or per thread *and* core or on system-wide fashion. The thread scope can be used to reduce the noise introduced by the rest of the processes running simultaneously on the system, since the monitoring events are attached only to the application and the VM threads. In addition, a thread scope can be combined with a core scope resulting in monitoring *a specific thread on a specific core*. However, there are some Hardware Performance Counters that cannot be utilized per thread, such as those dedicated to the memory controllers. Consequently, a per core scope is necessary. PerfUtil supports those features by exposing these different options via the PerfUtil API. The complete list of methods that form the PerfUtil API is shown in Listing 4.2.

4.2.3 Profiling With PerfUtil

A new `-XX` VM option has been introduced in MaxineVM that flags whether PerfUtil should be initialized or not. This way PerfUtil does not affect MaxineVM when it is not used. The `-XX:+UsePerfUtil` option initializes the PerfUtil module instance during MaxineVM start up, and consequently it allows the API utilization. However, the actual utilization of PerfUtil is controlled by the API calls. A VM engineer is responsible for injecting the API calls of choice into the proper VM call sites according to the profiling needs. This way, a VM engineer has fine control over when to start or stop the measurements at runtime, e.g., PerfUtil can start measuring after X full Garbage Collections, or Y method compilations, etc.

An abstract overview of how the PerfUtil API is deployed in the context of the current work is illustrated as an example use-case in Figure 4.3. The horizontal dimension of the figure provides a time-wise view of the different phases of the execution when the MaxineVM executes a Dacapo/Renaissance application [BGH+06,

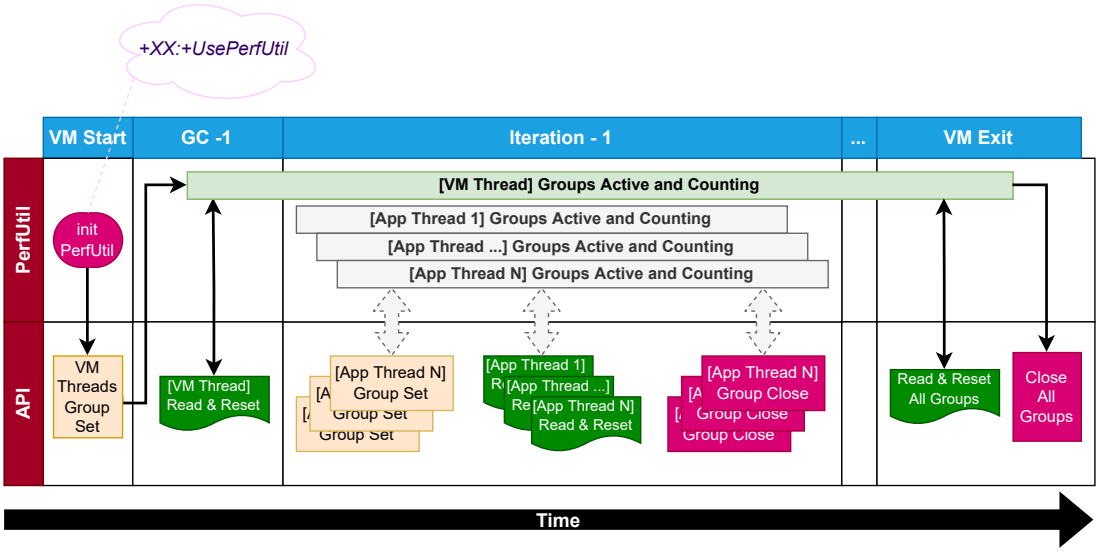


Figure 4.3: Example Use-Case: VM engineer - PerfUtil API interoperability & PerfUtil LifeCycle.

PRL+19][1]. The vertical dimension highlights the utilized API calls in relation to the layer/component where they have been injected by the user. The groups of choice are utilized with the "specific thread and any core" scope to profile per thread. Moreover, the results are gathered per explicit GC due to the iterative nature of Dacapo and Renaissance benchmark suites. The groups for the VM-internal threads are initialized (set) during the VM initialization phase since those threads are spawned once and they never close (only at VM exit). However, the groups of choice targeting the application threads are set during each mutation phase when each application thread is spawned. An application thread may close earlier than the whole mutation. It may even re-spawn multiple times thereafter, due to the multiple roles that can be assigned to a thread by the application. For that reason, a `PerfGroupReadAndReset` operation is performed at the closing of each thread. Finally, a `PerfGroupReadAndReset` operation is performed for the remaining threads/groups after mutation and before any explicit GC.

It is clear that PerfUtil is a flexible tool that offers a wide range of profiling capabilities. Through its easy-to-use API, the VM engineers are enabled to utilize Hardware Performance Counters and tailor them in accordance with their requirements.

[1]Those applications perform repetitive iterations, and a `System.gc()` call precedes each iteration.

4.2.4 Multiplexing

Each Perf group is composed by a set of "events of interest" that are simultaneously monitored for the same period of time in order for their results to be directly comparable. This is achieved only if the group is not "supernumerary" which means that there is at least one available and compatible Hardware Performance Counter for each event of the group. When this condition is not satisfied the whole group is rejected and none of the events is actually measured. Considering that modern CPUs typically have 8 general-purpose Hardware Performance Counters [Cor11a, AMD20], this complication is usual. Alternatively, the supernumerary group could be split to acceptable groups (i.e., with less than 8 events each) in order to get all events finally counted. However, by the time a group reserves all the available Hardware Performance Counters, the rest of the groups still will be getting rejected. As a result, complementary runs of the same application are needed to accommodate a different group each time. The above approach is called "Trace Alignment" [LCCAS14, NDP17]. Such a workaround was initially attempted by the current work due the simplicity it provides, but later it was put aside due to the following reasons. The Trace Alignment significantly increases the experimental time due to the complementary executions it requires in order to monitor all the events (i.e., the exeperimental is increased by 3x if the supernumerary group has been split to three subgroups). In addition, the final results might contain unaligned values of semantically related events (e.g. more llc misses than llc accesses). As a result, the approach of Trace Alignment was rejected and the so-called *time multiplexing* [LCCAS14, DEKB16] was leveraged instead for PerfUtil.

Alternatively, the Linux kernel is able to use time multiplexing for Perf events with a switch frequency of 100-1000 Hz [EGMdB15b] to enable multiple events to be counted by one physical counter. This technique breaks the barrier of the physical counters unavailability and enables a large set of events to be simultaneously counted. As a result, time multiplexing can be exploited to avoid the misaligned profiling values and simplify the experimental process by requiring only one experimental run. The count of an event corresponds only to the fraction of time that the event is actually counted; thus its value needs to be scaled to 100% (see Section 5.2.3). Consequently, multiplexing measurements might be less accurate (as in any sampling-alike technique) but it results in a trade-off between usability and accuracy. In the next paragraph, we measure the accuracy of PerfUtil when using multiplexing, and discuss whether lower accuracy comes at a high cost.

Table 4.1: Accuracy of PerfUtil with Multiplexing.

Metric	AVG	MEDIAN	Metric	AVG	MEDIAN
L1d Reads	-1.41%	-0.26%	CPU Cycles	0.01%	-0.44%
L1d Writes	-1.19%	-0.17%	Instructions	-1.78%	-0.23%
LLC Reads	-2.86%	-1.78%	Branch Instr	-1.79%	-0.42%
LLC Writes	-1.11%	-0.02%	LLC Miss Rate	-0.05%	0.02%
LLC Read Misses	-0.53%	1.02%	LLC Miss Rate (NUMA)	0.26%	0.24%
LLC Write Misses	-0.22%	-0.17%	MPKI	0.01	0.00
Node Read Misses	-21.78%	-3.54%	MPKI (NUMA)	0.00	0.00
Node Write Misses	-10.11%	-0.64%	Remote Mem Accesses	-4.79%	-0.45%
			Total AVG & MEDIAN	**-2.90%**	**-0.20%**

4.2.5 Accuracy of PerfUtil

The accuracy of PerfUtil was initially measured by comparing the profiling results of a MaxineVM build with PerfUtil without multiplexing in "specific thread on specific core" mode (`Maxine_PerfUtil_noMux`) against the output of the `perf stat` command attached to a MaxineVM build without PerfUtil (`Maxine_perf_stat`). After normalizing the results of `Maxine_PerfUtil_noMux` to those of the baseline (`Maxine_perf_stat`) the difference varied from -0.71% up to -1.23% and considered as sufficient to prove the validity of the implementation of PerfUtil.

However, for the reasons discussed in the previous section, it is clear that the employment of multiplexing might result to lower accuracy. As a result, we also evaluate and present in Table 4.1, the accuracy of multiplexing. For this experiment, the baseline is the `Maxine_PerfUtil_noMux` which has already been validated. Therefore, the profiling results of a MaxineVM build with PerfUtil and multiplexing in "specific thread on specific core" mode (`Maxine_PerfUtil_Mux`) are compared against the `Maxine_PerfUtil_noMux`. All values are normalized to the baseline. The results in this table indicate that the `Maxine_PerfUtil_Mux` yields accurate results in general because most of the metrics show $< 3\%$ average absolute difference. Benchmarks that show very small counts (according to the `Maxine_PerfUtil_noMux` and `Maxine_perf_stat`), such as the single-threaded *fop* negatively affect some average values (such as Node Read Misses and Node Write Misses), thereby leading to high differences (i.e., -12.78%, -10.11%). Counting those applications in a sampling manner might introduce considerable uncertainty because the events are sparse. Such a fact implies that the multiplexing technique may be inconsistent for applications that contain event counts below a threshold. Even though this threshold has not been explicitly defined yet for the PerfUtil implementation, the experimental data show that

it is approximately close to 2-3 millions counts of an event. In addition, it should be noted that the counts with lower values than this approximate threshold are observed only in events related to remote NUMA node accesses and only in single-threaded applications. Consequently, the inaccurate values make insignificant difference in observing the qualitative behavior of those applications because a single-threaded application normally tends to settle up on one NUMA node.

4.3 NUMAProfiler

4.3.1 Motivation

As explained in Section 4.1 the ultimate goal of the proposed research platform is to provide the means for evaluating whether a NUMA architecture can be beneficial for a managed application. The necessity of a tool such as NUMAProfiler arises from the blind spots that usually occur when the analysis considers only the low-level hardware metrics [PZFK20]. Although the metrics provided by PerfUtil are tightly coupled with overall performance observations, they lack of correlation with the application characteristics (i.e., the amount of data dependencies between the application threads). Scalability and performance, in the context of NUMA, is strongly affected by higher-level factors such as serial code sections, contention on shared resources, data locality and load balancing [PH90]. However, a profiling tool limited to the low-level hw metrics fails in sufficiently covering metrics related to all the above properties, but data locality. As a result, another tool that focuses on application metrics that reflect the degree of parallelism, balance, and the data dependencies is needed.

NUMAProfiler aims to enrich the tool-chain's insights with a higher-level point of view. It probes into crucial components of the JVM in order to monitor object allocations and object accesses per thread. An application profile at this layer of the stack is capable of providing insights regarding the memory allocation and thread data management patterns.

4.3.2 Overview

NUMAProfiler is a NUMA-aware Java object profiler. It aims to bridge the gap between the low-level hardware counters and high-level properties of applications. NUMAProfiler is a new component implemented into the runtime layer of MaxineVM, as illustrated in Figure 4.4. It profiles object allocations and accesses, and

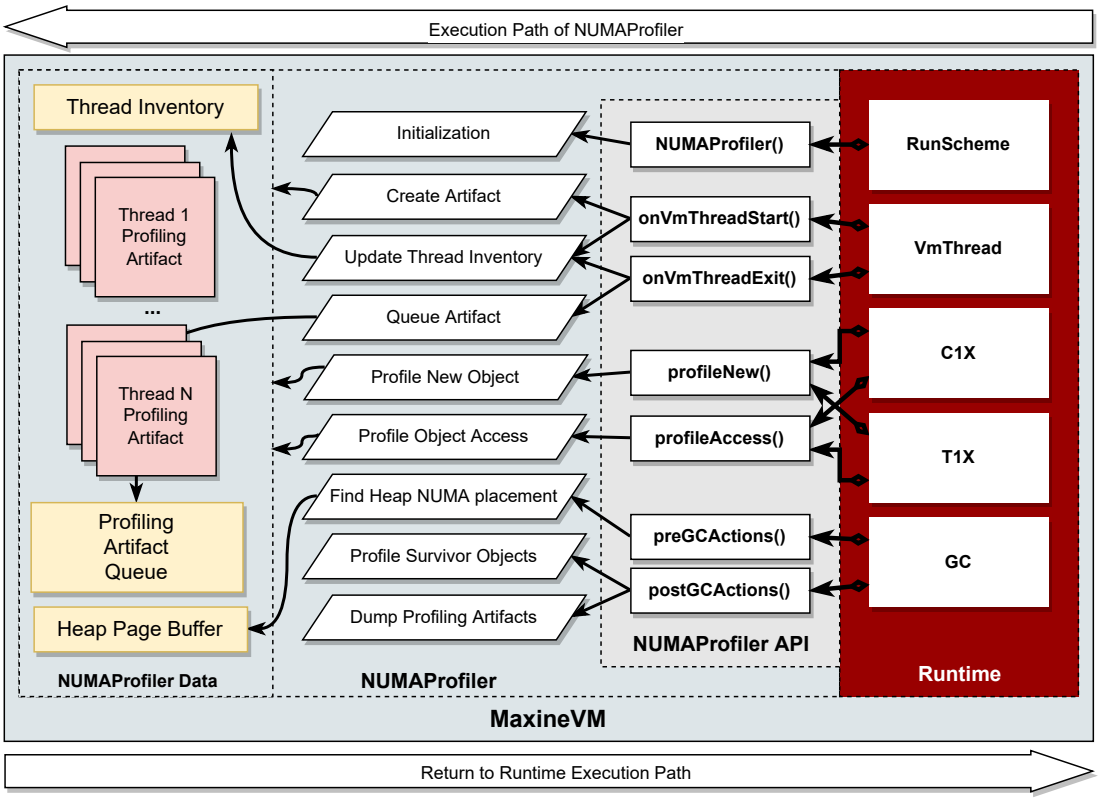


Figure 4.4: NUMAProfiler Overview.

it can produce the applications' thread map, heap trace, while also interfacing with the Linux NUMA library (i.e., libnuma [Ker10]). Moreover, NUMAProfiler takes advantage of the thread map and libnuma and classifies the object accesses as per "ownership" (shared/thread-local, see Section 4.3.7) and NUMA affinity (local/remote). NUMAProfiler exposes an API to VM. The API calls have been injected into the proper components of the Runtime and the Execution Engine of MaxineVM, and they lazily trigger the profiler mechanisms when it is necessary. Note that keeping the profiling overhead low is challenging, and inevitably, it has a major effect on design decisions undertaken which are discussed in the following sections.

NUMAProfiler is essentially a Java object allocated into the MaxineVM Java heap. It is initialized during the starting phase of MaxineVM, and it implements all the necessary internal functionality along with some auxiliary objects. It monitors object allocations, object accesses, locality of allocations and accesses, survivor objects after garbage collection, threads as well as the heap's virtual pages NUMA placement. Both the object allocation and the object accesses profiling functionalities are triggered upon their occurrence. The survivor object profiling and profiling output dump are

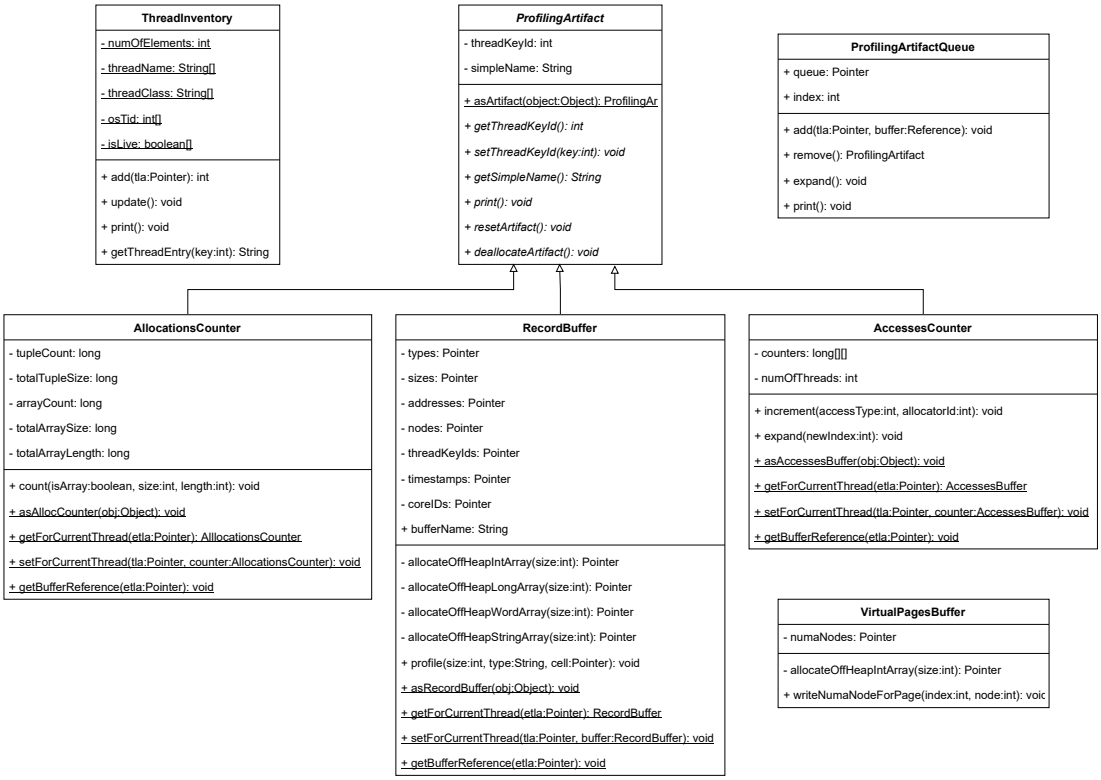


Figure 4.5: UML Diagram of profiling data structures.

performed in the post-GC phase. The collected data are stored in multiple data structures, as illustrated in Figure 4.5. Moreover, the profiling data are directly correlated with the application threads. The following Sections 4.3.3 to 4.3.8 present in detail the features of the NUMAProfiler. Throughout these sections the major design decision and challenges are reflected and explained.

4.3.3 Profiling Functionalities and Profiling Data Management

All the collected profiling data are stored using buffering techniques. The different buffer types of NUMAProfiler are illustrated in the UML diagram of Figure 4.5. As can be observed, NUMAProfiler contains multiple types of buffers because each one supports a different functionality; hence, each class reflects a different profiling data format:

- **Object allocation profiling:** refers to the profiling of each new object allocation. It is performed in two alternative modes: i) report the count of the allocated objects or, ii) trace and report each allocation in detail (object type, size, NUMA node, etc.). The first mode is supported by the `AllocationsCounter` and

the second by the `RecordBuffer`. The detailed profiling of object allocations comes with higher overhead and produces larger output files. Nevertheless, the decision regarding which profiling mode will be used is taken in accordance with the profiling needs. The first mode (i) is the default however, the `RecordBuffer` can also be utilized instead of the `AllocationsCounter` by using the `-XX:+NUMAProfilerTraceAllocations` option of MaxineVM in order to produce a detailed trace of object allocations.

- **Object access profiling:** refers to the profiling of the accesses that are performed by the VM and application threads to objects. For this functionality the `AccessesCounter` is used. The `AccessesCounter` buffer contains a 2-d array (the `counters[][]`) which stores the count of object accesses. Moreover, monitoring the object accesses enables the profiling of shared accesses (see Section 4.3.7).

- **Thread monitoring:** refers to the monitoring of the running VM and application threads. The `Thread Inventory` tracks each thread and is able to provide a timeline of the application's execution. Apart from that, the inventory assigns a profiler-internal unique ID to each thread instance (MaxineVM-internal thread IDs are re-used so two different thread instances might share the same ID). This ID is used by the profiler to characterize the access to an object as "thread-local" or "shared" (see Section 4.3.7).

- **Heap placement monitoring:** the `VirtualPages Buffer` keeps a track of the NUMA node that each virtual memory page of the heap resides (see Section 4.3.8). Therefore, the NUMA node that an object is placed onto can be found via the `VirtualPages Buffer`.

As can be observed in Figure 4.5 each class reflects a different profiling data format.

The `AllocationsCounter`, the `RecordBuffer`, and the `AccessesCounter` are subclasses of the `ProfilingArtifact` to enhance the modularity and flexibility of the design. To summarize, the `AllocationsCounter`, `RecordBuffer`, `AllocationsCounter`, `ThreadInventory`, `VirtualPagesBuffer` are used to store any type of profiling data. The buffering approach aims to gather the profiling data in memory as they are being collected and forward them to output in a controlled batched manner (i.e. one `ProfilingArtifact` as a whole) that minimizes the application execution interrupts.

4.3.4 Thread-Local Buffers

The design of the NUMAProfiler is thread-safe to minimize the profiler-derived blocking of the application threads. Each `ProfilingArtifact` instance is thread-local (its reference is stored in a MaxineVM-internal `VmThreadLocal` variable). As a result, the profiling buffers are not centralized; hence, there is no need of thread synchronization during profiling that would increase the profiling execution path overhead.

Dumping a `ProfilingArtifact` to the output file of NUMAProfiler blocks and distorts execution if it is performed during the mutation phase. However, a thread-local `ProfilingArtifact` follows the life cycle of its host thread. Therefore, in case a thread dies during mutation, the `ProfilingArtifact` will not be accessible thereafter. For that reason, a queue (*ProfilingArtifactQueue*) is used to store and maintain the `ProfilingArtifact` reference of any dead thread until the mutation phase ends. In the next post-GC phase, the `ProfilingArtifact` is sent to the output of NUMAProfiler; thus and the distortion of the mutation phase is avoided.

4.3.5 Off-Heap Profiling Data

Implementing a tool into the runtime layer of a metacircular VM inevitably results in additional Java objects in the heap derived by the tool itself. In case those objects are large (such as a `RecordBuffer`) they might end up triggering GC more often. Such a situation would be another form of distortion for the normal execution of an application. NUMAProfiler avoids this abnormality by leveraging the native memory management mechanisms of MaxineVM which enable off-heap object allocation for the profiling data buffers that contain extensively long arrays. More specifically, the `RecordBuffer` contains seven extensively long arrays (one element per allocation in each array, usually $\sim$ tens of millions), while the `VirtualPagesBuffer` contains one very long array (one element per virtual memory page of the heap; the larger the heap, the lengthier the array). As depicted in Figure 4.5, the attributes of those classes are of `Pointer` type (a MaxineVM-internal data type for a virtual memory address pointer). The `allocateOffHeap*` methods of those classes implement the off-heap memory allocation and return a `Pointer` of the allocated memory space which is then casted to the desired data type.

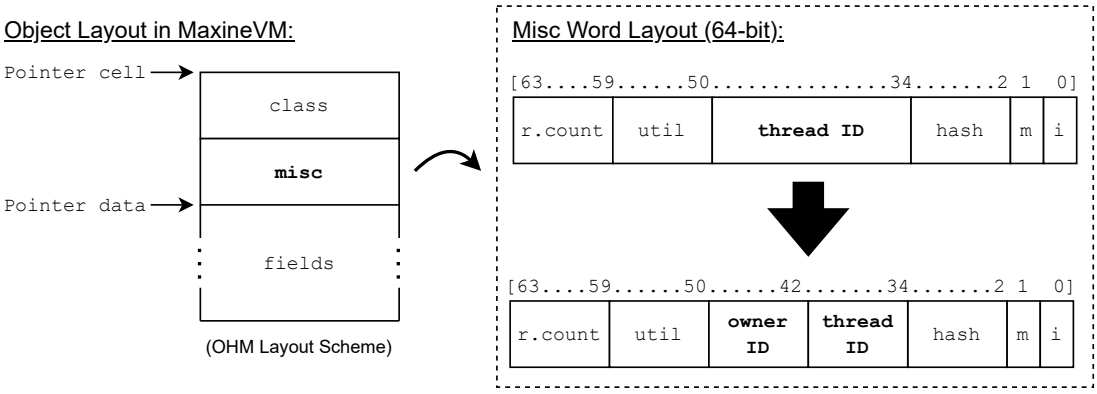


Figure 4.6: "Owner" thread ID injection into the Object Layout.

4.3.6 Interoperability with the MaxineVM Runtime

The "events" of object allocation and object access are modelled as code snippets in the JIT compiler (C1X) and interpreter (T1X) of MaxineVM. The API calls of NUMAProfiler have been injected into the proper snippets in order to be included into the execution code path when NUMAProfiler is enabled. This way, profiling mechanisms are lazily called when an event of interest occurs avoiding additional complexity. Although the injected profiling path inevitably comes with overhead, JVM's ordinary behavior is not heavily distorted (e.g. as with event-based interruptions). Moreover, profiling data are available at runtime thereby enabling their on-the-fly exploitation. The latter would be impossible or would come with excessive overhead if other techniques, such as instrumentation, were used.

4.3.7 Shared Object Accesses

The profiling of object accesses is rather trivial. However, classifying the profiled access as shared or thread-local is not because such information is not typically provided by the VM. Nevertheless, such a classification is important for characterizing the behavior of an application in the context of NUMA. A considerable amount of shared accesses indicates data dependencies between the application threads. This is very likelly to damage the locality of data if the dependent threads are scheduled on different NUMA nodes.

An object access is considered as shared if the thread that performed the access and the "owner" thread have different `ThreadInventory` IDs. Regarding NUMAProfiler, MaxineVM is modified to store the thread "owner" of each object into the misc word of the object header, as illustrated in Figure 4.6. As a result, the "owner" is available

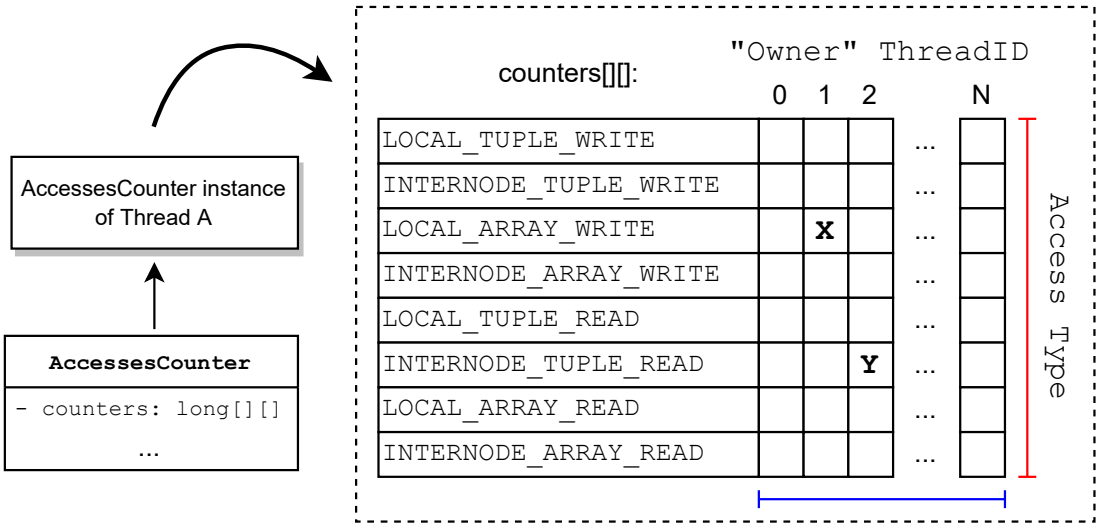


Figure 4.7: The format of the `counters[][]` array.

during the profiling of the access on an object along with the thread that performs the access. This information is stored to the `counters` 2-D array of `AccessesCounter` as illustrated in Figure 4.7. The vertical dimension index of the `counters` corresponds to the object access type, while the horizontal dimension index corresponds to the "owner" of the accessed object. Considering that each `AccessesCounter` buffer instance is dedicated to the thread that performs the access, the example of Figure 4.7 denotes that: *the thread A performed X `LOCAL_ARRAY_WRITES` to objects "owned" by thread 1, and Y `INTERNODE_TUPLE_READS` to objects "owned" by thread 2*[2].

In addition, Figure 4.7 highlights the different object access types (read or write, tuple or array, local or internode). An object access is characterized as remote in case the object is placed on a different NUMA node than the thread that performs the access (according to the `VirtualPages Buffer`).

Note that, the notion of the "owner" thread is quite abstract (i.e., can be the allocator thread, the last writer thread, or something else); therefore, it depends on the utilized context. Current work analyzes shared accesses considering the allocator thread (in Sections 5.3.1 and 5.4.1) and the last writer thread (in Chapter 6) as the "owner". The latter definition seems more reasonable in the context of NUMA.

[2]Thread A is the thread that this `AccessesCounter` instance is dedicated to.

4.3.8 NUMA-awareness

NUMA-awareness is enabled by the implementation of the `NUMALib` util which interfaces the MaxineVM with the NUMA library of Linux (libnuma) [Ker10]. The utilization of NUMALib by the NUMAProfiler enables the access to the native system calls of libnuma, such as `numa_move_pages()` (can return the NUMA node of a virtual memory address), `numa_node_of_cpu()` etc. In the pre-GC phase, NUMAProfiler finds on which NUMA node each heap virtual memory page resides and stores that information in the `VirtualPageBuffer`. Thereafter, the NUMA node of each object can be found by looking up for the address of the object in the `VirtualPageBuffer`. A hash method that obtains the memory page from the object address is used for the look-up, consequently the NUMA node of an object is found with O(1) complexity. Such a functionality essentially monitors the object placement of an application across the NUMA nodes. In addition, the `VirtualPageBuffer` capabilities are used in order to characterize an object access as local or remote.

4.3.9 NUMAProfiler's Accuracy

Table 4.2: NUMAProfiler's Accuracy.

(a) Allocations

Benchmark	Difference	Benchmark	Difference
avrora	0.28%	lusearch	-1.62%
fop	1.28%	pmd	2.04%
h2	-0.21%	sunflow	0.00%
jython	-0.04%	xalan	0.1%
luindex	-5.45%	**AVG**	**-0.4%**

(b) Accesses

Access Type	Difference
Read Tuple	2.63%
Read Array	0.03%
Write Tuple	0.05%
Write Array	0.04%
AVG	**0.69%**

Table 4.2 presents the accuracy of NUMAProfiler as measured for the current work. We quantify the accuracy of NUMAProfiler for the two major functionalities: object allocations and object accesses profiling. The count of object allocations using NUMAProfiler is compared against AntTracks [LBM15], a Java profiler for HotSpot VM using Dacapo 9.12 MR1 benchmarks. AntTracks has been used by [LBMW17] to measure the object allocations in Dacapo applications. However, it should be noted that a straightforward comparison against a MaxineVM profiler would be unfair. AntTracks has been developed for HotSpot VM which supports numerous optimizations related to object allocations, such as Escape Analysis (EA) and Compressed Object Pointers (Compressed Oops) while MaxineVM does not. EA allocates objects into the thread

stack instead of heap in case the object never "escapes" the allocator thread's scope resulting in lower object allocation counts. Compressed Oops reduce the application memory footprint by using less bits to represent the object pointers. Therefore, we need to rerun the experiments of [LBMW17] by disabling those optimizations in order to set the same baseline for both platforms. In addition, note that MaxineVM, as a metacircular VM written in Java, inevitably allocates some internal objects which HotSpot VM does not. We have excluded those objects from the results to isolate and validate the functionalities of NUMAProfiler[3]. The exclusion has been manually applied in a post-execution manner by removing any allocated object of any type that belongs to Maxine packages (`com.sun.max` etc.) from the profiling output. Table 4.2a presents the difference between the object allocations count of NUMAProfiler and AntTracks for all Dacapo applications. As can be observed the difference varies from -5.45% to 2.04%. Those differences are attributed to the no-deterministic behavior of the applications and/or to the fact that the comparison takes place between two different profiling implementations on two different VMs. Nevertheless, NUMAProfiler deviates from AntTracks only by -0.4% on average.

Unfortunately, AntTracks does not support profiling of object accesses; hence there is no point-of-reference for this functionality. For this reason, we implemented RWMicroBench, a custom Java benchmark with configurable and predictable count of object access events. We run RWMicroBench tuned to perform a fixed amount of object accesses of choice (Read Tuple, Write Array, etc.) on MaxineVM with NUMAProfiler enabled and compare the results against those expected. Table 4.2b presents the difference between the expected count of object accesses and the one reported by NUMAProfiler for each object access type. As can be observed the difference varies from 0.03% to 2.63%. Note that only Tuple Reads are slightly different from the expected number. This is attributed to read accesses performed to MaxineVM-internal objects. On average, NUMAProfiler reports only 0.69% deviation from the expected number.

[3]The manual exclusion of MaxineVM objects has been applied only for validation purposes because such a practice would be improper when the objective shifts towards studying a managed application hosted by a metacircular JVM. The ever-increasing popularity of metacircularity, as GraalVM and Truffle adoption denote, implies the significance of such an approach which is arising into a mainstream alternative. Consequently, unless otherwise stated, from that point onwards no discrimination is made between Application and JVM-internal objects.

4.3.10 Overhead Analysis of NUMAProfiler

As can be observed in Figure 4.4, the execution path of NUMAProfiler is injected into the execution path of the VM via the API calls. Therefore, the utilization of NUMAProfiler introduces additional overhead to the execution time of the application. This is an inevitable reality, even though the profiling data structures are thread-local, and the application threads do need additional synchronization. Upon the hit of a NUMAProfiler API call by an application thread, the JVM normal execution path temporarily stops for this thread until the API call returns. As a result, the more frequent and rapid the utilization of NUMAProfiler functionality is, the more additional overhead is introduced. The most expensive functionality is the profiling of object accesses since it occurs almost $\sim$100x more than object allocations. The overall execution time for Dacapo and Renaissance applications is increased by $\sim$ 5-20x when NUMAProfiler is enabled. Such a cost highlights the physical constraints of the "Java object profiling from the Runtime layer" technique. Calling the NUMAProfiler API calls less frequently, in a sampling manner, could potentially drop the overhead in lower levels. This is a direction for future work because the current work can tolerate this profiling overhead since NUMAProfiler is used only to provide the raw profiling results before the offline analysis of the applications (see Section 5.2). However, it is clear that currently, NUMAProfiler is not a "best-for-all" tool (i.e. not efficient to drive online optimizations).

To summarize, the numerous and novel profiling features, the flexibility, the modular and extendable design, and the programmability that NUMAProfiler offers suggest this profiler as an easy-to-use research tool. Finally, in case the overhead of NUMAProfiler would significantly be reduced (i.e., with sampling), this tool could potentially be utilized also as a testbed for quick optimization prototyping in the context of NUMA.

Chapter 5

Memory Characterization of Managed Applications

5.1 Introduction

In this chapter, the memory behavior of the Dacapo and Renaissance applications is studied. To conduct such a study the PerfUtil and NUMAProfiler tools are exploited. The study is presented per benchmarking in order to provide a suite-wide view to the reader. More specifically, Section 5.3 focuses on Dacapo benchmarks while Section 5.4 focuses on Renaissance. Prior to presenting the study, a detailed description of the experimental methodology is provided in Section 5.2. It is composed of the deployed machine characteristics, a brief overview of the utilized benchmarking suites and applications and, the experimental process.

5.2 Measurement Methodology

This section presents our methodology for studying the memory behavior of a managed application. The experimental process involves several Java applications executed on MaxineVM running on a two-socket machine.

5.2.1 Experimental Machine

Table 5.1 highlights the hardware and software setup of the machine, while Table 5.2 describes the utilized configuration. As can be observed, the utilized machine is essentially a NUMA system (Dell PowerEdge R620) with two nodes. To study the

Table 5.1: 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

Table 5.2: Run Configuration

	Single Node
Num of CPUs	1
Num of Available Cores	8
Num of Utilized Cores	8
LLC Size (MB)	20
Memory Controllers	4
DRAM Size (GB)	192
Java Heap Size (GB)	100
HyperThreading	off
Page migration	off

memory behavior of the applications any NUMA-related effect that might influence performance should be exluded. Therefore, only one socket and its corresponding DRAMs are deployed (the so-called *Single Node* configuration). The *Single Node* configuration encompass a Uniform Memory Access (UMA) environment; hence achieves that goal. As a result, the *Single Node* configuration provides 8 CPU cores and 192 GB of memory.

The dynamic behavior of the OS scheduler introduces uncertainty regarding an experimental process that involves performance measurements. Each thread might be assigned to a different CPU core, thereby affecting i.e., the locality of data in private caches, prefetching, etc. However, a precise assessment of the OS impact on managed applications is out of the scope of this thesis. Nevertheless, the metrics presented in Sections 5.3 and 5.4 and chapter 6 derive by the average of multiple measurements (see Section 5.2.3) in order to mitigate skewness. In addition, when having a NUMA machine, the scheduler should be configured appropriately to avoid unintentional thread migration in remote NUMA nodes. To that extent, the aforementioned Single

Node run configuration is a necessity. Moreover, HyperThreading is disabled to prevent additional performance and behavior variations. Similarly, the performance governor of the ACPI CPU frequency driver has been used to stabilize CPU frequency at 2.9 GHz to avoid dynamic scaling (Dynamic Voltage and Frequency Scaling - DVFS).

Note that, MaxineVM is utilized with the (default) SemiSpace Garbage Collector. Garbage Collection typically affects memory behavior because apart from collecting the "dead" objects in order to reclaim memory space, it moves the "live" objects around. The SemiSpace collector that is used in this work is a "stop-the-world" algorithm and the heap is composed by two discrete spaces, the "from" and the "to" space [JHM11]. Upon a collection, the live objects are moved from "from" space to "to" space; therefore such a behavior might affect locality of the application, especially on a NUMA machine (see Section 1.2). This work focuses on the memory behavior of the application during mutation and aims to exclude the effect of GC; as a result the effect of GC on memory behavior should be minimized. For that reason, the VM is configured with 100 GB of heap size (50 GB for each space) for all applications as can be seen in Table 5.2. This choice is based on the observation that none of the utilized applications allocates more than 40 GB of heap space. Consequently, 50 GB of "from" space will not trigger any implicit collection due to lack of free space during mutation. Moreover, both spaces fit in one NUMA node; hence even in the inevitable case of an explicit GC call by an application (`System.gc()`) during mutation the "live" objects will not be moved to another NUMA node. As a result, the effect of GC on an application's locality is minimized while it is assured that no side-effect of the NUMA system kicks in unintentionally.

5.2.2 Benchmarks Overview

A collection of Dacapo [BGH+06] and Renaissance [PRL+19] benchmarks are utilized for this study. Dacapo is a traditionally popular and well known benchmarking suite for Java, introduced in 2006, however it has evolved since then. It intends to reflect real world workloads that non-trivially exercise memory, thus its contributors have updated and modernized the included applications throughout the years. We use the latest pre-built maintenance release version (dacapo 9.12 MR1) [Che18] as provided by its authors. According to the release notes, it introduces `lusearch-fix` which comes with a fix in lucene and "dramatically changes the performance" of the application by "reducing the amount of allocation greatly" [Che18].

Table 5.3: Utilized Applications Overview.

	Benchmark	Input Size	Iterations	Benchmark	Input Size	Iterations
Dacapo	avrora	large (max)	30	lusearch	large (max)	30
	fop	default (max)	50	lusearch-fix	large (max)	30
	h2	huge (max)	20	pmd	large (max)	30
	jython	large (max)	30	sunflow	large (max)	30
	luindex	default (max)	50	xalan	large (max)	30
Renaissance	akka-uct	N/A	34	mnemonics	N/A	26
	reactors	N/A	20	par-mnemonics	N/A	26
	als	N/A	40	scrabble	N/A	60
	chi-square	N/A	70	neo4j-analytics	N/A	30
	gauss-mix	N/A	50	rx-scrabble	N/A	90
	log-regression	N/A	30	dotty	N/A	50
	movie-lens	N/A	30	scala-doku	N/A	20
	naive-bayes	N/A	40	scala-kmeans	N/A	60
	db-shootout	N/A	26	philosophers	N/A	40
	fj-kmeans	N/A	40	scala-stm-bench7	N/A	70
	future-genetic	N/A	60			

The Renaissance benchmark suite was introduced in 2019. To the best of our knowledge, most of the applications in Renaissance have not been a subject of a memory characterization study yet. It includes concurrent and parallel workloads hosted by modern Java frameworks, such as Akka [Inc09], Reactors.IO [IO13], Apache Spark [ZXW+16], Java Fork/Join [Cor14b], Java Streams [Cor14c], ScalaSTM [Tea10], and in-memory databases (Neo4j [Neo10], MapDB [Kot], ChronicleMap [Ope], MvStore [Gro05]). We use the pre-built 0.11.0 release[1].

Unfortunately, some applications from both suites were excluded due to incopatibility with either the MaxineVM or with a tool. For example, `batik` is incompatible with Java 8, `eclipse` spawns more thread instances than NUMAProfiler is able to support, while `tradebeans` and `tradesoap` are timeout-based, hence cannot run with NUMAProfiler due to the overhead introduced by the latter.

Table 5.3 lists all the utilized applications from both suites along with their run configurations. Well known practices for selecting the number of iterations for each application in order to reach a "run-steady" state [LBMW17, PRL+19] were used as a guide. The number of iterations that these works report ensures that the performance variations due to JIT compilation have been eliminated before the n-th iteration (warm-up). Hence, the application is in a run-steady state during the n-th iteration. However, the number of iterations used by the current work was augmented by 10 to

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

Table 5.4: Raw events/metrics collected by PerfUtil and NUMAProfiler.

PerfUtil Events	CPU_CYCLES	L1D_WRITES	NODE_PREFETCH
	RETIRED_INSTRUCTIONS	L1D_READ_MISSES	NODE_PREFETCH_MISSES
	BRANCH_INSTRUCTIONS	L1D_WRITE_MISSES	UNCORE_IMC_0_CPU_0
	BRANCH_MISSES	LLC_READS	UNCORE_IMC_1_CPU_0
	DTLB_READS	LLC_WRITES	UNCORE_IMC_2_CPU_0
	DTLB_WRITES	LLC_READ_MISSES	UNCORE_IMC_3_CPU_0
	DTLB_READ_MISSES	LLC_WRITE_MISSES	UNCORE_IMC_0_CPU_1
	DTLB_WRITE_MISSES	NODE_READ_MISSES	UNCORE_IMC_1_CPU_1
	L1_READS	NODE_WRITE_MISSES	UNCORE_IMC_2_CPU_1
	L1_WRITES	LLC_PREFETCH	UNCORE_IMC_3_CPU_1
	L1D_READS	LLC_PREFETCH_MISSES	
NUMAProfiler	TUPLE_ALLOCATIONS	TUPLE_READ_REMOTE	ARRAY_WRITE_REMOTE
	TUPLE_TOTAL_SIZE	TUPLE_WRITE_LOCAL	APPLICATION_THREAD_NAME
	ARRAY_ALLOCATIONS	TUPLE_WRITE_REMOTE	APPLICATION_THREAD_TYPE
	ARRAY_TOTAL_SIZE	ARRAY_READ_LOCAL	APPLICATION_THREAD_START_TIME
	ARRAY_TOTAL_LENGTH	ARRAY_READ_REMOTE	APPLICATION_THREAD_END_TIME
	TUPLE_READ_LOCAL	ARRAY_WRITE_LOCAL	

include enough run-steady iterations. Moreover, Dacapo applications allow the user to configure the input size and deployed threads with some exceptions (i.e. `avrora`) where thread number is determined by the input size. Renaissance applications have a "test" (small) and a "jmh" (default/large) input size and most of the benchmarks aim to automatically deploy worker threads equal to the number of available cores. The largest input size was used along with 8 threads, wherever possible.

5.2.3 Experimental Process

The experiments for the memory behavior characterization study were conducted in a two-step process in order to profile each application with each tool individually (PerfUtil/NUMAProfiler). This strategy was selected in order to keep the profiling "noise", that derives by the tool itself, as low as possible. Note that, a profiling process that deploys both tools concurrently is also possible, however additional logic is required in order to correlate the output traces in a time-wise manner. Nevertheless, the enhanced level of detail of such a feature is not required for the objectives of the current work (i.e., to measure the total object allocations, the LLC Misses per kilo Instructions, etc.).

For each step an individual build of MaxineVM equipped only with the corresponding tool was deployed. This strategy was selected in order to shield the profiling results, as possible, by the "noise" that each tool inevitably introduces. More

specifically, the first step of the experimental process deploys a MaxineVM build equipped with PerfUtil (MaxineVM_PerfUtil) to collect multiple microarchitectural metrics. Note that, by taking advantage of the multiplexing feature of PerfUtil (see Section 4.2.4) it was possible to concurrently monitor thirty-two (32) individual Perf events in a single run. The second step deploys another MaxineVM build equipped with NUMAProfiler (MaxineVM_NUMAProfiler) to collect various object-related metrics.

The collected raw events/metrics are highlighted in Table 5.4. All events/metrics are measured per thread and per iteration of the application. Note that multiple NUMA-related events/metrics exist in the table. Most of them are used in Chapter 6; nevertheless all metrics utilized in this thesis are listed here for consistency. The most events/metrics are common, and their meaning can be understood from their name. However, there are some rare and vendor-specific (UNCORE_IMC_<M>_CPU_<N>) or with ambiguous meaning (NODE_<READ/WRITE>_MISSES from PerfUtil or TUPLE_READ_REMOTE from NUMAProfiler). Moreover, some events with rather clear naming (such as the LLC_<READ/WRITE>_MISSES) need additional explanation in the context of NUMA to avoid confusion. A list of those events follows along with an explanation for each one:

- LLC_<READS/WRITES>: all L2 accesses that missed the L2 and finally served by LLC.
- LLC_<READ/WRITE>_MISSES: all LLC accesses that missed in any **local** or **remote** LLC and finally served by **local or remote** memory.
- NODE_<READ/WRITE>_MISSES: all LLC misses that missed in any **local** or **remote** LLC and finally served by **remote** memory.
- LLC_PREFETCH_MISSES: all L2 hardware prefetch requests that served by **local or remote** memory.
- NODE_PREFETCH_MISSES: all L2 hardware prefetch requests that served by **remote** memory.
- UNCORE_IMC_<M>_CPU_<N>: all read and write requests to the integrated memory controller M of the NUMA node N.
- <TUPLE/ARRAY>_<READ/WRITE>_REMOTE: all read/write accesses on tuple objects or arrays that were placed in the remote NUMA node.

The collected raw events/metrics are processed offline after the experimental procedure, in order to calculate the synthetic metrics that are presented in Sections 5.3 and 5.4, and Chapter 6. Such an example is the `LLC MISS RATE` = `LLC_READ_MISSES/LLC_READS`. Metric values that refer to an application as a whole (i.e., total object accesses) are calculated in a *coarse-grained* manner, as the average result of the ten last run-steady iterations. Metrics presented in a per-thread basis (i.e., object accesses of thread N) refer only to the last run-steady iteration because the threads are re-instantiated per iteration in many applications obstructing correlation across iterations.

5.3 Dacapo

Dacapo[2] is one of the most popular benchmark suites in the context of managed applications; and Java in particular. It has been widely used in numerous workload characterization studies over the years [KMJV12, DBSEE13, LBMW17] as well as in validating new profiling tools [DBSEE13, LBM15] and evaluating proposed optimizations [GTSS13, GTS+15]. Similarly, this work has employed Dacapo applications to validate and measure the accuracy of NUMAProfiler (see Section 4.3.9).

Throughout the NUMAProfiler validation process, a number of observations have been found that complement existing works with respect to Dacapo characterization. For example, [LBMW17] characterizes `sunflow` as memory intensive when observing high-level metrics, such as allocation rate and allocation size. However, the current work finds that this observation does not hold true when cache and physical memory pressure metrics are taken under consideration, due to low contention and good data locality `sunflow` exhibits. Moreover, Kalibera et al. [KMJV12], and Lengauer et al. [LBMW17] do not assess the latest Dacapo maintenance release (9.12-bach-MR1) [Che18] which is used in this thesis. Therefore, the updated versions of Dacapo applications along with the addition of lusearch-fix [Che18], to the best of our knowledge, have not been yet characterized in the literature.

This section presents a study over the memory behavior of the Dacapo benchmarks. PerfUtil and NUMAProfiler are leveraged to conduct this study, while the findings aim to complement those presented by the current state-of-the-art studies in regard to the Dacapo benchmark suite [KMJV12, DBSEE13, LBMW17]. Nevertheless, the

[2] https://github.com/dacapobench/dacapobench

proposed methodology can be applied to any application running on the JVM.

Note that in Section 4.3.9, we manually excluded the MaxineVM-internal objects in order to fairly compare the NUMAProfiler with AntTracks. Therefore, this practice was leveraged only for validation purposes. The ever-increasing popularity and adoption of GraalVM and Truffle implies the significance of metacircularity which is arising into a mainstream alternative. Consequently, when the objective shifts towards studying a managed application that is executed by a metacircular compiler or VM the exclusion of component/VM-internal objects would be an improper practice in the sense that it obfuscates the reality. For that reason no discrimination is made between objects that belong to the application or to MaxineVM from that point onwards.

5.3.1 Object Metrics

Tables 5.5 and 5.6 outline the object allocation and access characteristics for each Dacapo application. More specifically, Table 5.5 presents the total number of object allocations, the allocation sizes in MBs along with the allocation rate in objects and size per second. In addition, some object layout properties are discussed, such as the average object size, etc. Table 5.6 presents the total number of object accesses, the access rate, and Read/Write (R/W) ratio. Moreover, it shows the amount of shared object accesses for each application as a percentage of the total accesses. An object access is considered as "shared" in case the "accessor" thread is different from the thread that allocated the object. The numbers are calculated by averaging the metrics achieved by ten iterations beyond the warm-up phase (see Section 5.2.2). The maximum value of each metric is highlighted as bold in both tables.

Object Allocations

`H2` allocates the most objects and memory in total, however it is one of the less allocation-intensive applications based on the allocation rate. This is due to the large number of instructions (and consequently execution time) that `h2` has, as shown in Table 5.7. `Sunflow` allocates less and smaller objects but turns out to be the most allocation intensive application in both terms of objects and memory size. `Jython` is the most intensive single-threaded benchmark both in terms of objects and memory size allocation. `Lusearch-fix` has been introduced as an update to `lusearch` bearing a fix in the lucene platform that reduces object allocations; however, no such difference

Table 5.5: Object Allocations and Object Layout in Dacapo.

	Object Allocations				Object Layout		
Application	10^3	MB	$\frac{10^3}{sec}$	$\frac{MB}{sec}$	Avg obj size [b]	Array Rate [%]	Avg Array Length
avrora	9,320	406.7	419.9	18.3	45.8	22.1%	11.2
fop	2,783	158.5	4,711.3	268.4	59.7	31.8%	19.9
h2	**461,524**	**24,762.6**	920.6	49.4	56.3	33.9%	12.5
jython	210,975	16,987.4	4,593.2	369.8	84.4	28.0%	57.3
luindex	187	25.8	194.4	26.8	**144.6**	**39.3%**	**202.4**
lusearch	30,787	3,876.4	12,718.6	1,601.4	132.0	24.8%	118.8
lusearch-fix	30,786	3,876.4	12,730.0	1,602.9	132.0	24.8%	118.8
pmd	27,380	1,541.3	3,886.9	218.8	59.0	23.0%	31.8
sunflow	175,755	7,344.0	**43,248.9**	**1,807.2**	43.8	2.6%	3.0
xalan	113,709	13,024.8	11,318.1	1,296.4	120.1	36.0%	56.9
GEOMEAN	**27,474**	**2,085.7**	**3,715.1**	**281.9**	**79.6**	**22.6%**	**34.7**

is observed[3].

Memory Footprint & Object Layout

Total object allocations do not necessarily reflect the allocation size footprint (per iteration, in MB). `Luindex` allocates the largest objects on average, and it contains the most and longest arrays. However, it is a single-threaded application with the smallest memory footprint among Dacapos. `Lusearch` and `xalan` follow in terms of average object size showing also higher array rate and average array length than the geometric mean of Dacapos. `Xalan` is an application that has large objects, on average. Even though it performs fewer allocations than `sunflow`, it ends up having higher memory footprint. Consequently, such metrics are crucial especially when the observed memory footprint origin matters (i.e., GC optimizations, an optimization targeting large objects, heap size tuning and more). Moreover, the Object Layout metrics reveal additional properties of an application which are very likely to affect its memory and/or overall behavior. For example, large objects (as in `lusearch` and `xalan`) are more likely to span across two memory pages. Such applications might stress TLB and page-table more than others. This type of memory pressure can potentially be a source of inefficiencies and severe overheads in the context of NUMA [GLD+14, APB+20]. For example, `lusearch` and `xalan` have been proven

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

Table 5.6: Object Accesses in Dacapo.

	Object Accesses			Sh. Accesses	
Application	10^6	$\frac{10^6}{sec}$	R/W Ratio	R	W
avrora	4,865	219	10:1	93%	57%
fop	65	109	9:1	0%	0%
h2	51,518	103	16:1	41%	31%
jython	7,518	164	10:1	0%	0%
luindex	134	139	13:1	0%	0%
lusearch	2,357	974	15:1	24%	4%
lusearch-fix	2,362	977	14:1	24%	5%
pmd	1,144	162	10:1	36%	7%
sunflow	7,504	1,847	30:1	84%	0%
xalan	7,433	740	12:1	19%	23%
GEOMEAN	**2,378**	**322**	**13:1**	**12%**	**4%**

unfriendly to the Page Migration mechanism of Linux [PZFK20] which is tightly related to the TLB and page-table. `Lusearch` slows down by ∼13% while `xalan` gets its remote node accesses increased by ∼300% when Page Migration is enabled.

Object Accesses

The object accesses highlight the application-memory relation degree as observed from the application layer. Table 5.6 presents a collection of object accesses metrics as well as the percentage of shared accesses. The latter refers to the number of accesses performed by a different thread rather than the "owner" of the object (see Section 4.3.7) as a percentage of total object accesses. As can be observed in Table 5.6, `h2` performs the most object accesses in total while, `sunflow`, `xalan`, and `avrora` have more than 2x more accesses than the geomean. `Sunflow` performs the most object accesses per second followed by `lusearch` and `xalan`. All applications are read-dominated with `sunflow` having the most (30) reads per one write. `Avrora` shows the highest shared object R/W access rate. `Sunflow` follows, but it has only shared Read accesses(see last column of Table 5.6). A high percentage of shared Read accesses is an indication for the existence of the producer-consumer pattern. Finally, `avrora`, `h2`, and `xalan` show the highest percentage of shared Writes accesses.

Object Metrics Summary

To summarize, we group the applications by allocation and access rate and filter out those below the geometric mean using the heuristic of Equation (5.1):

$$(\ Allocation\ Rate > Geomean\ )\ \textbf{AND}\ (\ Access\ Rate > Geomean\ ) \tag{5.1}$$

Consequently, according *only* to the object metrics studied so far, the applications that seem to stress the memory system the most are: `lusearch`, `luserach-fix`, `sunflow`, and `xalan`.

The above results confirm the already known trends [KMJV12, LBMW17] even though they slightly differ in terms of absolute numbers. This differentiation can be a result of the "metacircular JVM effect", the slightly different run configurations, and/or the different (updated) version of the benchmark suite.

In contrast to the literature [KMJV12, LBMW17], we observe that although the above metrics are necessary, they are not sufficient to properly characterize the memory behavior of a managed application. Those metrics comprise only an application layer point of view, and thus it is impossible to assess the application's effect and interaction with the underlying hardware components, such as the LLC. For that reason, we extend the study by co-examining some microarchitectural metrics provided by PerfUtil.

5.3.2 Hardware Metrics

Hardware Instructions Overview

Table 5.7 shows the distribution of Arithmetic (integer and floating point), Branch, and Memory Instructions per benchmark. Furthermore, the collected metrics are also presented as a percentage over the total number of retired instructions. Such a table format can reveal the ratio between Memory, Arithmetic, and Branch Instructions as well as whether an application is potentially dominated by read or write accesses. PerfUtil is able to count the *Total Retired Instructions*, *L1D Reads*, *L1D Writes*, *Branch Instructions* and consequently the number of *Arithmetic Instructions* is calculated as:

$$Arithmetic = Total - L1DReads - L1DWrites - Branch \tag{5.2}$$

Read and write ratios settle towards read operations, since the L1D read instruction percentage is higher than L1D writes in all applications. It tends to be aligned to

Table 5.7: HW Instructions Overview - Dacapo

Application	Instructions	L1D Reads	L1D Writes	Total Mem.	Arith.	Branch
avrora	57,645,363,283	35%	18%	56%	29%	15%
fop	2,644,295,397	28%	25%	47%	38%	15%
h2	1,292,129,013,772	29%	16%	48%	37%	15%
jython	294,432,673,937	27%	32%	43%	40%	16%
luindex	5,085,147,760	29%	25%	43%	40%	18%
lusearch	72,686,381,316	28%	21%	43%	40%	17%
lusearch-fix	72,517,120,274	28%	21%	43%	40%	17%
pmd	36,072,750,432	28%	22%	46%	38%	16%
sunflow	162,257,371,836	31%	21%	43%	43%	15%
xalan	223,669,834,459	29%	21%	48%	36%	15%
GEOMEAN	**67,741,600,233**	**29%**	**22%**	**46%**	**38%**	**16%**

Table 5.8: Cache/Memory Locality and Pressure in Dacapo.

Application	CPI	BPU MPKI	Misses PKI				Miss Rate				Accesses PK Object Operation	
			DTLB	L1	L2	LLC	DTLB	L1	L2	LLC	LLC	Memory
avrora	**1.21**	**4.36**	**1.62**	**24.28**	**12.72**	0.13	0.3%	**4.4%**	**52.4%**	1.0%	153	2
fop	0.76	1.01	0.92	13.21	5.85	**1.16**	0.2%	2.8%	44.3%	19.8%	**233**	**46**
h2	**1.17**	**2.66**	**1.81**	12.37	**7.16**	**2.60**	**0.4%**	2.6%	**57.9%**	**36.3%**	178	**65**
jython	0.54	0.46	0.44	8.75	2.23	**0.98**	0.1%	2.0%	25.5%	**44.2%**	82	**36**
luindex	0.55	2.00	0.37	5.24	1.87	0.13	0.1%	1.2%	35.7%	6.8%	70	6
lusearch	0.74	1.64	**1.20**	**16.29**	**6.19**	**0.91**	0.3%	**3.8%**	38.0%	14.6%	**189**	**28**
lusearch-fix	0.72	1.64	1.18	16.10	6.16	0.91	0.3%	**3.8%**	38.2%	14.8%	**188**	**28**
pmd	0.78	1.83	**2.01**	**17.17**	**7.36**	0.83	**0.4%**	**3.7%**	**42.9%**	11.2%	**229**	26
sunflow	0.59	**2.50**	0.54	7.55	2.30	0.72	0.1%	1.8%	30.5%	**31.3%**	48	15
xalan	**0.94**	1.60	1.04	**19.59**	6.17	**0.92**	0.2%	**4.0%**	31.5%	14.9%	**181**	27
GEOMEAN	**0.77**	**1.71**	**0.96**	**12.83**	**4.95**	**0.68**	**0.21%**	**2.8%**	**38.6%**	**13.8%**	**138**	**19**

the R/W Ratio of Table 5.5, i.e. `sunflow` has the largest gap. However, minor misalignments are visible probably due to the "noise" introduced by the complex infrastructure of the VM. Note that, for example, the observed total instructions of a managed application are essentially a mix of instructions from the application **and** the VM itself. Since there is no obvious way to safely estimate and exclude the latter, a co-interpretation of the results derived from the NUMAProfiler and PerfUtil is necessary.

Even though it is clear that memory instructions outweigh branch and arithmetic instructions in all applications, a deeper examination of metrics related to the memory hierarchy is needed in order to characterize the memory behavior.

Data Locality & Cache/Memory Pressure

Although the cache hierarchy aims to fill the latency gap between the CPU and main memory, the latter often remains a source of delays in a program's execution. Due to complex features of modern hardware (e.g., out-of-order execution, multiple cache levels, shared memory, etc.) such characterization lacks of a strict definition (or concrete methodology) and can only rely on a multi-aspected profile that comprises numerous metrics. However, CPI co-examination along with memory hierarchy and Branch Prediction Unit (BPU) *pressure* and *locality* metrics can reveal useful insights regarding memory behavior. **Misses Per Kilo Instructions (MPKI)** can be used as a global metric of "pressure" because it factors in the total retired instructions [ZBF10, MG11]. On the other hand, **miss rate** of each memory level can provide an indication of "data locality". In addition, **LLC and memory accesses per kilo object operations** (allocations + accesses) aim to bridge low with high level metrics and are quite indicative regarding data locality. Moreover, note that large pressure on BPU derives from non-predictive control flow, or data-dependent branches, or both. A non-predictive control flow leads in accessing memory locations in a non-deterministic manner, thereby influencing the regularity of the memory access pattern. Additionally, in the case of data-dependent branches, the accessed memory location cannot be predicted leading to irregular memory access patterns. Therefore, a large value of BPU MPKI can indicate irregularities in the memory access patterns which in the case of a memory intensive application will penalize performance. The following sections discuss the above metrics which are listed in Table 5.8 per application. The maximum value of each metric is highlighted as bold.

The larger the MPKI value is, the "heavier" the load for the corresponding memory hierarchy level is. Consequently, this set of "pressure" metrics can be used to assess the memory-bound degree relatively with other applications. LLC MPKI reveals that an application's object allocation and access intensiveness are not necessarily reflected in main memory pressure which is counter-intuitive. For instance, `sunflow` is the most intensive application in terms of object allocations and object accesses; however, the largest pressure on main memory among the Dacapo benchmarks is caused by `h2`. On the contrary, `sunflow` seems to put the least pressure, among the multithreaded applications, on the memory hierarchy as the LLC/Memory pressure and CPI metrics. This is justified to the good spatial and/or temporal locality of `sunflow` working data. `Sunflow's` behavior can also be observed in LLC and memory accesses per kilo object operation ratio which are below the geomean and among the lowest. Consequently,

the accesses per kilo object operation metrics are quite indicative regarding the locality of working data by comparing object operations against actual cache/memory pressure. `Fop` is the most LLC and main memory intensive among the single-threaded applications (`fop`, `jython`, `luindex`), which is also confirmed by its CPI. It is notable that although `avrora` and `pmd` are the most LLC intensive applications, they finally put low pressure on memory denoting that their data set successfully fits into the larger LLC (compared to L2). After examining LLC and memory pressure metrics, `avrora's` CPI seems to be affected more by BPU, DTLB, and cache rather than main memory (Table 5.8). High DTLB pressure of `lusearch` and `xalan` probably is related to their large objects.

Summary of Dacapo Memory behavior analysis

Section 5.3.1 has discussed object related metrics while Section 5.3.2 and Section 5.3.2 have discussed hardware related metrics in a top-down manner. Section 5.3.1 concluded that the most memory intensive Dacapo applications are: `lusearch`, `luserach-fix`, `sunflow` and `xalan`. However, after co-examining the hardware related metrics in Section 5.3.2 this conclusion was finally overthrown. More specifically, `sunflow` hardly stress the LLC while `lusearch` and `xalan` put moderate pressure to LLC compared to `avrora`. The latter has among the lowest object allocation and access rates but turns out to be the most LLC intensive. `H2` is the most main memory intensive Dacapo application along with the single-threaded `fop`. `H2` is also an example of a memory-bound application. Counter-intuitively, `sunflow` turns out to be the least main memory intensive application; therefore stands as an exceptional example of high and low level metrics co-examination. Another notable finding is that `lusearch-fix` does not essentially differ from `lusearch`.

5.4 Renaissance

This section studies the memory behavior of the recently introduced Renaissance benchmark suite [PRL+19]. Similarly to the Section 5.3, several high and low-level memory related metrics derived from the NUMAProfiler and PerfUtil are presented and discussed in a top-down manner. The study aims to provide the research community with useful insights for this new benchmark suite which, to the best of our knowledge, are not yet available. Moreover, this section acts as an intermediate step before analysing how Renaissance benchmarks behave in the NUMA context.

Table 5.9: Object Allocations and Object Layout in Renaissance.

	Object Allocations				Object Layout		
Application	10^3	MB	$\frac{10^3}{sec}$	$\frac{MB}{sec}$	Avg obj size [b]	Array Rate [%]	Avg Array Length
akka-uct	**1,052,172**	**56,561**	69,027	3,710	56	2	9
reactors	387,087	15,942	6,562	270	43	18	13
als	62,361	2,671	7,666	328	45	5	146
chi-square	105,632	3,474	22,567	742	34	2	42
gauss-mix	457,453	13,180	61,284	1,767	30	1	43
log-regression	11,302	1,068	1,563	148	99	8	545
movie-lens	175,959	12,749	7,535	546	76	22	113
naive-bayes	561,601	12,917	**220,859**	**5,080**	24	0.05	96
db-shootout	429,679	36,004	29,539	2,475	88	51	59
fj-kmeans	17,051	18,844	1,311	1,448	**1,159**	**64**	225
future-genetic	40,305	2,285	6,219	352	59	4	69
mnemonics	260,821	12,709	7,515	366	51	18	17
par-mnemonics	261,017	12,724	9,189	448	51	18	17
scrabble	66,366	2,981	23,428	1,052	47	15	6
neo4j-analytics	524,089	17,751	29,762	1,008	36	4	12
rx-scrabble	10,877	452	7,199	299	44	5	9
dotty	48,554	1,726	7,076	252	37	6	66
scala-doku	174,394	4,177	17,165	411	25	1	10
scala-kmeans	6,017	194	3,537	114	34	0.11	**675**
philosophers	68,417	4,115	13,977	841	63	15	19
scala-stm-bench7	35,024	1,724	11,753	578	52	14	22
GEOMEAN	**101,155**	**5,216**	**12,271**	**633**	**54**	**5**	**40**

5.4.1 Object Metrics

As already stated, object allocations, accesses, and their rates over time, are quite indicative of an application's memory intensiveness; however, they do not always lead to well-rounded conclusions as shown in the previous section. Table 5.9 presents metrics related to object allocations and object layout, while Table 5.10 presents metrics related to object accesses for each Renaissance application. The reported numbers are the average of ten iterations beyond the warm-up phase (see Section 5.2.2). The maximum value of each metric is highlighted in both tables (Tables 5.9 and 5.10).

Object Allocations

`Akka-uct`, `naive-bayes`, `neo4j-analytics`, `h2`, and `gauss-mix` allocate the most objects in total (per iteration). `Akka-uct` allocates almost double the objects of `naive-bayes` which is the second highest allocating application. `Mnemonics` and `scala-doku` are the single-threaded applications with the most total object allocations.

Memory Footprint & Object Layout

The number of total allocations do not necessarily reflect the allocation size footprint (per iteration, in MB). `Fj-kmeans` has by far the largest average object size (1.13 kB) while `log-regression`, `db-shootout`, and `movie-lens` follow. It is notable that although `fj-kmeans` and `db-shootout` allocate fewer objects than `naive-bayes` or `neo4j-analytics`, they end up with higher memory footprint which is apparently related with object size. In addition, `fj-kmeans` is an array-dominated application with 63.9% of its allocations being arrays while `db-shootout`, and `movie-lens` follow.

Object Allocation Rate

The aforementioned metrics indicate how much memory an application allocates, however, they do not highlight how intense the memory allocation is. The object count and object size per second metrics should be taken under consideration towards characterizing the memory intensity of a managed application. An application that allocates new objects at a high rate is very likely to put excessive pressure on the memory system. `Naive-bayes`, `akka-uct`, `gauss-mix`, `neo4j-analytics`, `db-shootout`, and `scrabble` are the most intensive in terms of both object and size allocation rate.

Object Accesses

Table 5.10 presents a collection of object accesses metrics as well as the percentage of shared accesses. The latter refers to the number of accesses performed by a different thread rather than the "owner" of the object (see Section 4.3.7) as a percentage of total object accesses. `Akka-uct`, `fj-kmeans`, `reactors`, and `neo4j-analytics` perform by far the most accesses to their objects. `Akka-uct` and `fj-kmeans` show the highest object access rate along with `philosophers`, `neo4j-analytics`, `scrabble`, and `naive-bayes` that follow. All applications are dominated by read accesses, with

Table 5.10: Object Accesses in Renaissance.

	Object Accesses			Shared Accesses	
Application	10^6	$\frac{10^6}{sec}$	R/W Ratio	R	W
akka-uct	**23,126**	**1,517**	5:1	69%	3%
reactors	10,496	178	6:1	**77%**	**56%**
als	1,649	203	22:1	9%	16%
chi-square	1,000	214	8:1	27%	0%
gauss-mix	2,301	308	6:1	36%	0%
log-regression	1,984	274	110:1	2%	2%
movie-lens	4,391	188	10:1	34%	11%
naive-bayes	1,215	478	112:1	3%	2%
db-shootout	4,343	299	3:1	28%	6%
fj-kmeans	15,668	1,204	**120:1**	59%	1%
future-genetic	2,053	317	18:1	37%	5%
mnemonics	4,996	144	7:1	0%	0%
par-mnemonics	5,001	176	7:1	47%	0%
scrabble	1,411	498	8:1	47%	0%
neo4j-analytics	10,151	576	3:1	51%	1%
rx-scrabble	201	133	6:1	48%	0%
dotty	584	85	13:1	0%	0%
scala-doku	1,393	137	71:1	0%	0%
scala-kmeans	373	219	27:1	0%	0%
philosophers	3,802	777	6:1	60%	30%
scala-stm-bench7	894	300	9:1	37%	0%
GEOMEAN	**2,405**	**292**	**13:1**	**14%**	**0.4%**

`fj-kmeans` having 120 reads per one write. On the other hand, `db-shootout` and `neo4j-analytics` have the most balanced R/W ratio.

Many applications show a considerable degree of shared accesses with reactors having the most shared reads and writes. Even though actor frameworks aim to guarantee workload concurrency their asynchronous non-blocking message passing infrastructure inevitably leads to accessing objects "owned" (allocated) by other threads. Therefore, the degree of shared accesses for `reactors` and `akka-uct` is justified. Nevertheless, as will be discussed in Chapter 6, shared data might essentially block scalability in a NUMA architecture. On the contrary, `als`, `log-regression`, `naive-bayes` show negligible shared object accesses, thus potentially denoting data parallelism.

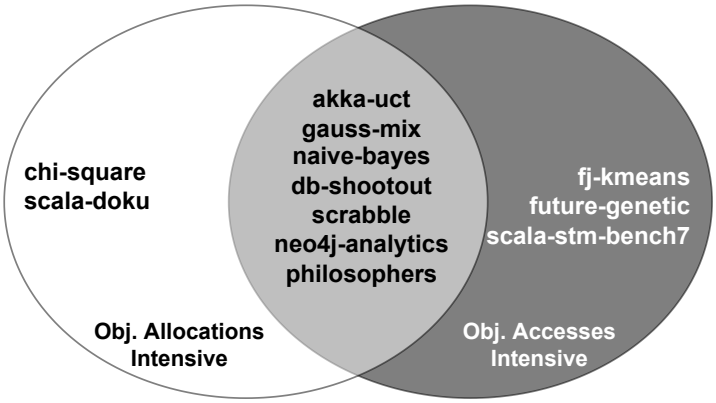


Figure 5.1: Memory Intensive Renaissance Applications in terms of Object Allocations and Accesses.

Object Metrics Summary

Up to this point we have surveyed several high-level memory metrics related to object allocations and object accesses. To summarize, we group the Renaissance applications by allocation and accesses rate and filter out those that are below the geometric mean, using the heuristic described in Equation (5.1). According *only* to the object-related metrics, the applications that seem to stress the memory system the most are depicted in Figure 5.1. The left circle contains applications with object allocation rate above the geometric mean (object allocation intensive). Similarly, the right circle contains applications with object access rate above the geometric mean (object access intensive). The intersection of the two circles highlights the applications that are intensive both in terms of object allocations and object accesses.

5.4.2 Hardware Metrics

Hardware Instructions Overview

Table 5.11 shows the distribution of arithmetic, branch and memory instructions per benchmark. Memory instructions tend to outweigh branch and arithmetic instructions in the vast majority of the applications, similarly to Dacapo. However, Renaissance applications show a greater diversity than Dacapo. For instance, `als`, `chi-square`, `movie-lens`, and `naive-bayes` (all belong to the Apache Spark family) are below the percentage of minimum memory instructions observed in Dacapo, while `future-genetic` is beyond the maximum one. Nevertheless, the geometric mean of the memory instruction % is, as expected, $\sim$ 45% and all applications are dominated

Table 5.11: Hardware Instructions Overview - Renaissance.

	Instructions	L1D Reads	L1D Writes	Total Mem.	Arith.	Branch
akka-uct	413,031,096,621	29.55%	15.66%	45.21%	36.04%	**18.76%**
reactors	344,901,909,214	30.71%	21.11%	51.82%	33.10%	15.08%
als	173,577,079,497	24.27%	8.80%	33.08%	**50.50%**	16.43%
chi-square	44,712,182,871	25.27%	14.85%	40.13%	42.21%	17.66%
gauss-mix	88,927,702,969	26.10%	16.23%	42.33%	40.15%	17.52%
log-regression	66,506,869,326	**35.13%**	8.43%	43.56%	38.08%	18.36%
movie-lens	255,423,610,313	25.35%	13.83%	39.18%	44.77%	16.05%
naive-bayes	95,767,198,846	27.76%	9.29%	37.05%	44.90%	18.05%
db-shootout	**415,462,411,962**	26.35%	16.76%	43.11%	41.17%	15.71%
fj-kmeans	16,587,430,766	32.52%	13.92%	46.44%	38.33%	15.23%
future-genetic	50,065,010,991	34.77%	22.75%	**57.52%**	29.78%	12.70%
mnemonics	167,497,623,059	27.12%	18.73%	45.86%	38.40%	15.74%
par-mnemonics	164,897,514,508	27.01%	18.87%	45.87%	38.39%	15.74%
scrabble	44,953,086,743	28.06%	19.48%	47.54%	37.09%	15.38%
neo4j-analytics	176,983,336,923	28.52%	15.90%	44.43%	39.41%	16.17%
rx-scrabble	6,552,607,394	31.60%	**24.72%**	56.32%	30.93%	12.75%
dotty	19,001,592,728	26.46%	15.72%	42.18%	40.98%	16.83%
scala-doku	48,122,342,431	29.73%	12.10%	41.83%	40.83%	17.34%
scala-kmeans	9,360,648,141	30.52%	17.90%	48.42%	38.50%	13.09%
philosophers	90,682,695,138	31.01%	20.21%	51.22%	33.63%	15.15%
scala-stm-bench7	21,856,846,101	28.14%	18.68%	46.83%	38.39%	14.79%
GEOMEAN	**72,999,411,837**	**28.71%**	**15.76%**	**44.87%**	**38.56%**	**15.84%**

by read accesses.

Data Locality & Cache/Memory Pressure

To further examine memory behavior of the Renaissance benchmarks we need to focus on Memory and Cache subsystem. Table 5.12 presents the same metrics that are studied for Dacapo applications as well (see Section 5.3.2). The maximum value of each metric is highlighted in Table 5.12.

As can be observed, `reactors`, `scrabble`, `dotty`, and `scala-stm-bench7` have high CPI values. Such a fact could imply stalls due to memory and consequently memory-boundness. However, this observation contradicts with Figure 5.1 where only `scrabble` and `scala-stm-bench7` seem to be "object allocation and accesses intensive". Therefore, it becomes clear again that the assessment of memory intensity cannot rely only on object-level metrics. An application might turn out to be memory bound as a result of other reasons (i.e., lack of memory locality), even though it does not significantly allocate or accesses objects. In particular, `reactors` shows 2.5x

Table 5.12: Cache/Memory Locality and Pressure in Renaissance.

	CPI	BPU MPKI	Misses PKI				Miss Rate				Accesses PK Object Operation	
			DTLB	L1	L2	LLC	DTLB	L1	L2	LLC	LLC	Memory
akka-uct	0.76	1.37	1.14	24.53	6.84	**2.50**	0.26%	5.52%	27.88%	36.57%	122	45
reactors	1.02	1.22	0.70	17.63	7.18	0.85	0.13%	3.38%	40.74%	11.85%	251	30
als	0.41	0.32	0.06	2.73	0.87	0.26	0.02%	0.82%	31.80%	29.64%	98	29
chi-square	0.62	0.67	0.08	9.45	2.32	1.20	0.02%	2.32%	24.59%	51.53%	95	49
gauss-mix	0.49	0.41	0.10	13.77	4.21	2.21	0.02%	3.22%	30.57%	52.46%	138	72
log-regression	0.60	2.38	0.11	4.87	1.02	0.56	0.02%	1.11%	20.98%	55.13%	35	19
movie-lens	0.61	1.17	0.38	8.88	3.21	0.86	0.10%	2.29%	36.11%	26.85%	204	55
naive-bayes	0.47	0.06	0.07	12.76	3.60	2.02	0.02%	3.41%	28.19%	56.28%	197	111
db-shootout	0.52	0.53	0.16	8.76	3.35	1.52	0.04%	2.08%	38.20%	45.52%	293	**133**
fj-kmeans	0.75	1.22	0.59	8.96	4.57	2.70	0.13%	1.91%	51.07%	**58.93%**	5	3
future-genetic	0.80	0.88	0.28	10.03	3.84	0.70	0.05%	1.80%	38.24%	18.29%	105	19
mnemonics	0.64	2.89	0.64	14.60	2.58	1.20	0.14%	3.14%	17.68%	46.57%	91	43
par-mnemonics	0.68	3.05	0.25	15.20	3.05	1.24	0.05%	3.27%	20.06%	40.75%	105	43
scrabble	**1.46**	**4.11**	1.05	25.29	4.79	1.05	0.22%	5.27%	18.93%	21.84%	162	35
neo4j-analytics	0.73	0.98	0.52	19.74	3.86	1.81	0.12%	4.44%	19.54%	47.02%	65	31
rx-scrabble	0.88	2.68	0.27	20.05	3.45	1.14	0.05%	3.52%	17.18%	33.07%	117	39
dotty	1.10	4.08	**2.24**	**29.16**	**9.34**	1.65	**0.52%**	**6.81%**	32.03%	17.64%	**311**	55
scala-doku	0.63	1.63	0.57	15.32	8.65	1.39	0.14%	3.62%	**56.47%**	16.08%	295	47
scala-kmeans	0.55	2.11	0.22	3.85	0.95	0.46	0.04%	0.78%	24.82%	47.92%	26	13
philosophers	0.85	1.70	0.63	15.43	3.63	0.37	0.13%	3.07%	23.54%	10.32%	92	10
scala-stm-bench7	1.07	1.61	1.07	21.01	7.00	2.10	0.23%	4.54%	33.30%	30.03%	169	51
GEOMEAN	**0.71**	**1.20**	**0.34**	**12.34**	**3.51**	**1.12**	**0.08%**	**2.74%**	**28.45%**	**31.92%**	**106**	**34**

more LLC accesses per kilo object operations than the geomean which implies lack of data locality, while `dotty` shows very high BPU MPKI which indicates irregularity in memory access patterns (see Section 5.3.2 that explains how the BPU MPKI is related to irregular memory access patterns). An additional remark is `par-mnemonics` which has been named as memory-bound [PRL+19], however its CPI is way below 1; hence, it turns out that memory is not the most decisive factor for performance of this application. An interesting finding occurs when focusing on `als` and `movie-lens` which have been classified as compute-bound [PRL+19]. This description is confirmed for `als` by the observed CPI value, and by the negligible pressure it puts on any level of the memory hierarchy. On the contrary, `movie-lens` lacks locality because it shows 2x more LLC accesses per kilo object operations than the geomean; hence, the performance of this application is rather influenced also by memory.

Moreover, `akka-uct`, `gauss-mix`, `naive-bayes`, `db-shootout`, `scrabble`, `neo4j-analytics`, and `philosophers` are in the intersection of Figure 5.1; hence it is expected to be memory intensive applications. Nevertheless, they diversify because not all put pressure on both the LLC and memory. `Akka-uct`, `gauss-mix`, `naive-bayes`, `db-shootout` and `neo4j-analytics` put significant pressure up to memory, as expected. On the contrary, `scrabble` and `philosophers` put significant pressure only up to the LLC. It is very likely that `scrabble` and `philosophers`

either benefit from locality in LLC or/and have a smaller working data set that fits into LLC. Although, we cannot safely estimate the exact reason for each, note that low LLC and memory accesses per kilo object operation of `philosophers` indicate good data locality. On the other hand, `scrabble` shows high BPU MKPI and as is described in Chapter 6 it lacks of data locality due to irregular memory access patterns. In addition, `rx-scrabble` (that implements the same algorithm as `scrabble` but uses an alternative framework), `dotty`, `mnemonics`, `par-mnemonics`, `scala-kmeans`, `philosophers`, and `log-regression` are candidates for irregular memory patterns due to their high BPU MPKI. It should be noted that an irregular application might avoid heavy penalization in a system with unified caches and create an "illusion" of locality in case the working data fits into the caches or by luck due to cached data co-location in the same (local) cache. However, its behavior might dramatically change in an architecture with distributed resources among nodes (such as NUMA) where not only working data might be spread in local and remote caches, but also cache coherency mechanisms might cross-invalidate them causing locality breakdown.

A similar diversity is observed in the right section of Figure 5.1 where the applications are intensive in terms of object accesses. `Fj-kmeans` and `scala-stm-bench7` put significant pressure on both LLC and memory, while `future-genetic` stresses only the LLC. `Fj-kmeans` is a notable case because it is the second most intensive application in terms of object accesses, it is a read-dominated application, and according to those data it seems to benefit from data locality since it has the lowest LLC accesses per kilo object operations. Note that this data locality may be fragile or not always guaranteed as will be discussed in Chapter 6. The "object allocations intensive" `chi-square` and `scala-doku` do not put significant pressure neither to LLC nor to Memory, which is expected since object allocation operations are 3-5 order of magnitude fewer than object accesses.

By comparing the overall geometric mean of Renaissance and Dacapo, it is observed that the former puts relatively lower pressure to LLC. Nevertheless, such a fact does not necessarily mean lower main memory pressure in which Renaissance gets over Dacapo. The data locality of the applications in the LLC is decisive for the overall memory intensiveness.

`Log-regression`, `naive-bayes`, `gauss-mix`, and `chi-square`, which are Spark applications, show greater than 50% LLC Miss Rate. Such a poor data locality inevitably brings to the spotlight the effect of the Spark engine when co-located with worker threads over the same limited CPU and cache resources. However,

only gauss-mix and naive-bayes end up with high memory pressure among the aforementioned applications. It is notable that akka-uct has lower CPI than reactors despite the fact that the first accesses main memory almost three times more. This counter-intuitive observation can be explained by the fact that reactors is more memory instruction dominated (see Table 5.11), and has 2x more LLC accesses per kilo object operations than akka-uct. Throughout this comparison, the high complexity of memory behavior analysis is again highlighted, thereby denoting that memory overhead might derive literally by any component of the stack.

Key Findings

Section 5.4.1 has discussed object related metrics, while Section 5.4.2 has discussed hardware related metrics and our key findings are summarized below:

1. The most intensive applications in terms of object allocations and accesses are: akka-uct, gauss-mix, db-shootout, fj-kmeans, future-genetic, scrabble, neo4j-analytics, philosophers, and scala-stm-bench7.

2. As derived by Table 5.12, the most LLC intensive applications are: dotty, scala-doku, akka-uct, reactors, scala-stm-bench7, fj-kmeans, and scrabble.

3. The most memory intensive applications are: akka-uct, fj-kmeans, gauss-mix, naive-bayes, scala-stm-bench7, neo4j-analytics, dotty, and scala-doku.

4. Similarly to the Dacapo applications, the Renaissance applications that present high number of object allocations and object accesses do not always put the highest pressure on LLC and memory. For instance, gauss-mix, db-shootout, future-genetic, and philosophers put moderate pressure on the LLC; even below the geometric mean. On the contrary, reactors, dotty, and scala-doku put relatively high pressure to the LLC, although they are not included in the aforementioned "object-intensive" application category. Regarding the pressure on main memory, akka-uct, fj-kmeans, gauss-mix, naive-bayes, scala-stm-bench7, neo4j-analytics, dotty, and db-shootout are the most intensive.

5. Even though actor frameworks aim to guarantee concurrency and non-blocking parallelism, the actor applications show a considerable amount of shared data.

Their actual effect is of significant interest towards studying those applications in the context of a NUMA architecture where data dependencies might bound scalability.

6. It is further confirmed that `als` is a compute-bound application in contrast with `movie-lens` which is affected by locality of data.

7. `Par-mnemonics` and `scrabble` do not directly confirm their memory-bound characterization.

5.5 Summary

To summarize, we have augmented the understanding about the memory behavior of both Dacapo and Renaissance benchmark suites. The two tools that were presented in Chapter 4 were deployed in order to collect a wide range of metrics. PerfUtil collected numerous low-level hardware metrics, while NUMAProfiler gathered high-level Java metrics related to object allocations and accesses (Section 5.2). Those metrics were utilized in order to study and analyse the memory behavior of 30 managed applications from the Dacapo and Renaissance benchmark suites. The study of the memory behavior was presented in two parts. Section 5.3 focused on Dacapo applications, while Section 5.4 focused on Renaissance applications. The findings from both Sections 5.3 and 5.4 showcase the effectiveness of the hw/sw co-utilization perspective in the profiling of a managed application. The next chapter (Chapter 6) will study several application properties towards characterizing the scalability of Dacapo and Renaissance application in a NUMA system.

Chapter 6

NUMA Scalability Analysis of Managed Applications

6.1 Introduction

In this chapter, the applications of the Dacapo and Renaissance benchmark suites are characterized in the context of a NUMA architecture. As explained in Chapter 2 the high latency of the remote memory access in a NUMA system might penalize performance. At the same time, the interference of cache coherency mechanisms might lead to cache/memory utilization inefficiencies (i.e., repetitive invalidation of cached data) due to various reasons, such as data dependencies, and/or inefficient scheduling decisions.

That being said, memory overheads are not the only concern. As Figure 1.1 shows, even compute-bound applications (such as `als`) might also be penalized by a NUMA system. Therefore, a characterization that also considers further scalability properties is essential since the performance in a NUMA system is mainly achieved through scalability (both in terms of threads and memory).

Application properties related to scalability, such as the amount of *serial code sections*, *contention on shared resources*, *data locality*, and *load balancing* [PH90] need to be assessed. The notion of "serial code sections" and "load balancing" is essentially the degree of parallelism that an application exhibits. They can be quantified by counting the number of exploited threads along with the workload balance. In case the multithreaded workload is not equally distributed (balanced) among the deployed threads (i.e., 90% of the load is processed by one thread), the thread count might be insufficient. Therefore, the thread count, and the balance of the

workload, act as complementary to each other towards characterizing the parallelism degree of an application.

In addition, the "contention on shared resources" essentially regards the data dependencies among the working threads. Assuming that an application is as parallel as needed in order to benefit from a NUMA system and the threads do not migrate across nodes, the shared data between multiple threads is the only factor that can potentially lead those threads to contend for the shared LLC slices and memory. Such a behavior might not only increase contention, but also negatively affect data locality and subsequently penalize performance or block scalability. Consequently, both data dependencies and locality is of significant interest towards characterizing the behavior of an application on a NUMA system.

As a first step, attributes related to parallelism (# of deployed threads, workload balance) are evaluated in Section 6.2. Thereafer, additional memory-related properties are also assessed for those applications with parallel workloads. The data dependencies among the deployed threads are assessed in Section 6.3 by taking under consideration the shared read and write object accesses. Finally, Section 6.4 empirically characterizes data locality by comparing the LLC locality between NUMA and non-NUMA executions of the applications.

As explained above, the next sections gradually assess the Dacapo and Renaissance applications per property towards a holistic NUMA characterization. Each property analysis results to a set of homogeneous application groups. Their intersection results to a ***NUMA characterization table*** where each application belongs to a single category. Such a result not only succeeds in characterizing Dacapo and Renaissance applications, but also sheds light into the existing application categories and their NUMA behavior. These research findings aim to augment our understanding regarding the required properties of managed applications to take advantage of NUMA and draw a conclusion under which circumstances NUMA can be beneficial.

6.2 Workload Parallelism & Balance

Table 6.1 presents a collection of metrics per application to evaluate parallelism and memory balance. 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 table (Table 6.1) breaks down the workload carried

Table 6.1: Parallelism and Balance of the Dacapo and Renaissance 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

out per thread type (Main, Aux, Workers). The workload of each type is expressed as a percentage over the total retired hardware instructions.

The workload carried out by worker threads is further analyzed under the scope of balance (last column of Table 6.1). The variables C, I, and OA in the last column of Table 6.1 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 (6.1) is used.

The Equation (6.1) calculates the **imbalance in X** between the N threads [DFF+13]. A high number of standard deviation can yield in high imbalance, assuming that the average value is constant (Imbalance = 0% means "totally balanced").

$$Imbalance\ in\ X = \frac{stdev(X_{thread1}, ..., X_{threadN})}{average(X_{thread1}, ..., X_{threadN})} \tag{6.1}$$

The imbalance of hardware instructions provides an indication regarding the overall imbalance of the workload. However, complexity and latency vary among hardware instructions, i.e., a memory instruction might stall the CPU for more cycles than an arithmetic instruction that uses the ALU. Therefore, it is necessary to also examine memory imbalance in order to evaluate the contribution of memory operations to the overall workload imbalance. The imbalance of object accesses is utilized in order to examine the memory imbalance of an application. The imbalance of object accesses is indicative regarding the computations-versus-memory heterogeneity of the worker threads[1]. Nevertheless, the imbalance of hardware instructions and of object accesses might not necessarily 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. For that reason we examine the impact of this heterogeneity on performance through the imbalance in CPU cycles in order to justify the actual degree of imbalance for an application.

The following sections characterize the properties of parallelism and balance for the Dacapo and Renaissance applications by examining the aforementioned metrics (# of workers, imbalance of hardware instructions, CPU cycles, and object accesses). This characterization step concludes in five discrete categories: the "Single-Threaded" (see Section 6.2.1), the "TLP-Bound" (see Section 6.2.2), the "Embarrassingly Imbalanced" (see Section 6.2.3), the "Imbalanced" (see Section 6.2.4), and the "Explicitly Parallel" (see Section 6.2.5). Figure 6.1 illustrates an overview of the aforementioned categories along with the examined metrics as a flow chart. All the categories, but the "Explicitly Parallel" indicate that the applications do not exhibit the degree of parallellism that is needed in order to efficiently scale on a NUMA system; hence are very likely to be parallelism-bound.

[1] Hereafter, when this thesis refers to the term "computation-memory heterogeneity", it refers to the computations-versus-memory heterogeneity.

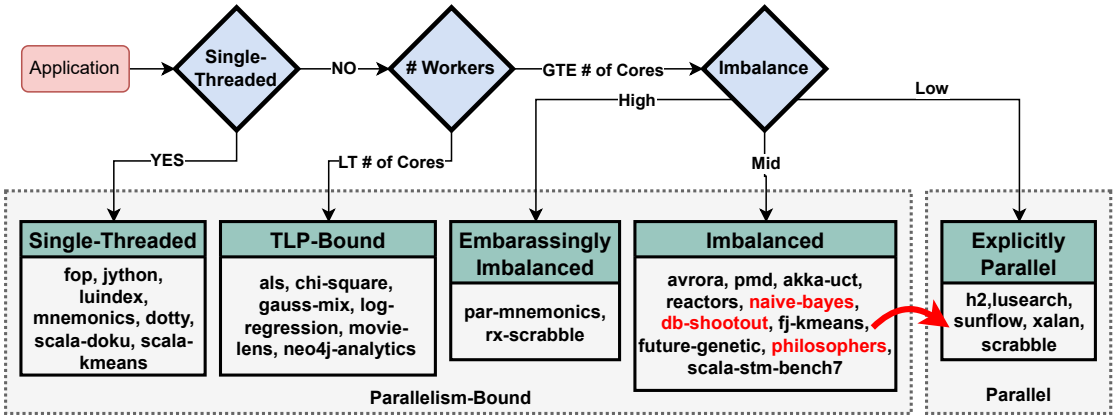


Figure 6.1: NUMA Scalability Characterization: Parallelism.

6.2.1 Single-Threaded

`Fop`, `jython`, `luindex`, `dotty`, `mnemonics`, `scala-doku` and `scala-kmeans` do not spawn parallel threads; hence thet are 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`).

6.2.2 TLP-Bound

If the number of deployed threads is lower than the available CPU threads, it is usually an indication that the Thread Level Parallelism (TLP) is below the capacity of the system. Hence, the scalability of the application is practically bound as it cannot effectively exploit the scale-out resources. `Als`, `chi-square`, `gauss-mix`, `log-regression`, `movie-lens`, and `neo4j-analytics` are such examples because they 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.

6.2.3 Embarrassingly Imbalanced

`Par-mnemonics` and `rx-scrabble` show extreme imbalance in hardware instructions. This fact is justified by the observation that only one worker thread processes 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.

6.2.4 Imbalanced

Considerable but lower imbalance in hardware instructions is observed also in other parallel applications, such as `avrora`, `pmd`, `akka-uct`, `reactors`, `naive-bayes`, `db-shootout`, `fj-kmeans`, `future-genetic`, `philosophers`, and `stm-bench7`. 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 lower correlation denotes that the effect of memory in the imbalance of CPU cycles varies across the applications.

Towards assessing the imbalance effect, we also take under consideration some key characteristics of the applications in order to conclude whether the observed computational and/or memory imbalance is actually harmful.

For example, `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.

`Pmd` analyzes multiple source code files in an imbalanced manner due to the unequal sizes of the input files [DBSEE13]. 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 to 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 [ZJ13] in the Akka actors framework. UCT processes a tree of tasks with variable size which are assigned to workers by the Akka dispatchers via a shared task queue structure [ZJ13]. Rosa et al. [RCB16] report non-uniform task distribution across actors which is also confirmed by the imbalance in hardware instructions of the worker threads (see Table 6.1). 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 [IS14] benchmarks that are implemented into the Reactors.IO framework [IO13]. 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, it is notable that 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. This fact denotes that 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 also falls into to the "Explicitly Parallel" category.

`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 kmeans algorithm using an 8-worker Fork-Join thread pool. The observed imbalance in CPU cycles that is also reported by Rosales et al. [RRB20] probably denotes either computation-memory heterogeneous sub-tasks

Table 6.2: Root cause.

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

or/and sub-tasks of unequal size. The imbalance in hardware instructions and object accesses in Table 6.1 indicate both. Sub-tasks of unequal size and the ineffectiveness of the work stealing in preserving the balance are rather counter-intuitive findings for a Fork-Join application.

Scala-stm-bench7 is single-threaded for $\sim$ 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 above analysis of the "Imbalanced" applications reveals that the observed imbalance is caused by various reasons and root causes. Table 6.2 summarizes the different root causes along the applications. Note that, the last two root causes might indicate that the application is not "Imbalanced".

6.2.5 Explicitly Parallel

Applications that deploy multiple (equal or greater than the available CPU cores) and balanced workers are considered as "Explicitly Parallel". H2, lusearch, sunflow, xalan, and scrabble fullfill both critera thus, they belong to the "Explicitly Parallel" category. However, it should be noted that the h2 has extensively serial phases. Even though all the deployed worker threads of this application are balanced, almost the 30% of its workload is carried out by the main thread.

6.2.6 Summary of Parallelism & Balance Characterization

This section has characterized the studied applications regarding parallelism and balance. The characterization along with the examined criteria are summarized

in Figure 6.1. Five discrete application classes seem to exist: the "Single-Threaded", the "TLP-Bound", the "Embarrassingly Imbalanced", the "Imbalanced", and the "Explicitly Parallel". The "Single-Threaded", "TLP-Bound", and "Embarrassingly Imbalanced" applications are *a priori* unable to scale in a NUMA system because they lack of Thread-Level Parallelism. Therefore, they cannot effectively exploit scale-out NUMA resources. In addition, the "Imbalanced" applications bear several inefficiencies such as, unequal task distribution, serial algorithm/code sections, and/or memory imbalance, which are very likely to prevent them from effectively exploiting scaled out NUMA hardware resources. Moreover, such patterns lead to imbalanced resource utilization which might create contention to the interconnect and finally block scalability [DFF+13]. On the contrary, the "Explicitly Parallel" applications effectively exhibit parallelism and balance. Hence, they are strong candidates for sufficient scaling on a NUMA system. Finally, the studied application set contains some ostensibly "Imbalanced" benchmarks that essentially are "Explicitly Parallel". This happens because they either concatenate multiple parallel and balanced workloads (i.e., `db-shootout`), or there is one "special" thread among many balanced worker threads (i.e., `naive-bayes, philosophers`).

6.3 Data Dependencies

Section 6.2 has classified the studied applications in terms of the exhibited parallelism and workload balance. However, parallelism and workload balance, though necessary, they are not sufficient conditions for NUMA scalability. The effect of factors related to memory can lead to substantial inefficiencies that also affect overall performance, and finally either offset scalability gains or even lead to performance degradation. The dependencies of data among the working threads is such a critical factor because it can potentially undermine the distribution of workload across the NUMA system. Ideally, the multiple worker threads that are spread across NUMA nodes should process their "own" data without sharing any data with other workers; especially with those running on a remote NUMA node. As will be explained in the following sections, an object that is being written by multiple threads that run on remote nodes will not only result in increasing the expensive remote node accesses but also in additional cache invalidations. Both results tend to increase the pressure on main memory and the interconnect traffic. This section evaluates the degree of data sharing between worker threads by examining the metric of shared object accesses that is provided by

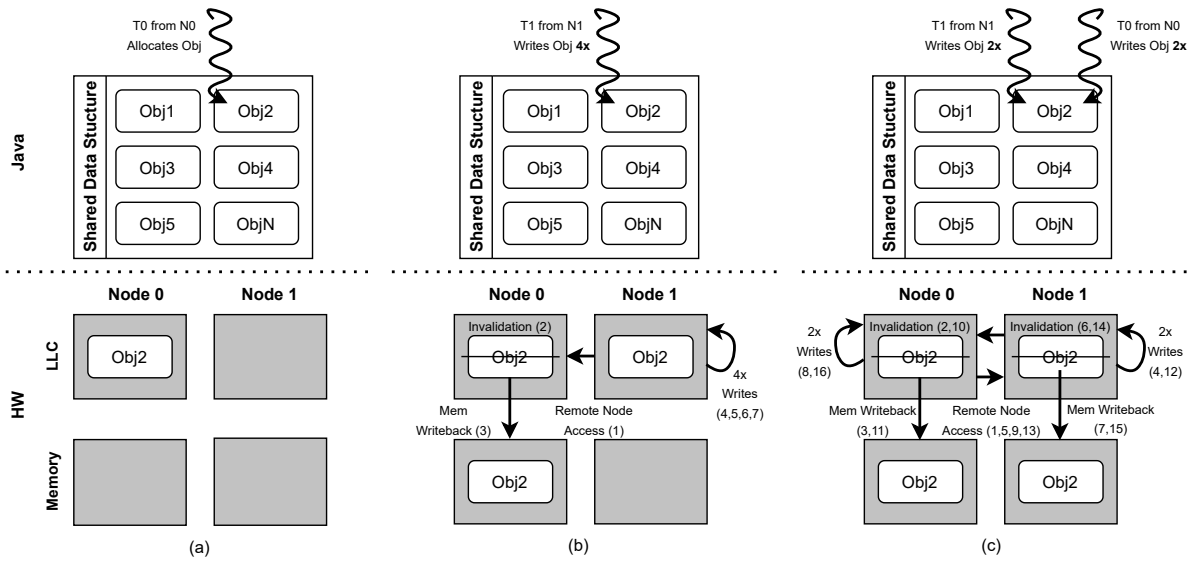


Figure 6.2: Effect of Write Object Access in NUMA.

NUMAProfiler for each application.

The amount of shared accesses of the Dacapo and Renaissance applications has been presented in Sections 5.3 and 5.4. These sections define the "owner" of an object as the thread that allocated the object. Nevertheless, such a heuristic would be insufficient in the context of a NUMA architecture. Figure 6.2 illustrates how write accesses are handled by a typical cache coherent NUMA system that implements the MESIF protocol. The Subfigure (a) shows the Obj2 (object 2) that is allocated by T0 (thread 0) on NUMA node 0 while, Subfigures (b) and (c) illustrate two different cases of 4 consecutive object writes. More specifically:

- Subfigure (b): four "shared" writes are performed by T1 which is running on NUMA node 1 (remote). Following the steps shown in parentheses, the first write operation triggers a remote node access to fetch the data from the LLC of node 0. According to the MESIF cache coherency protocol, the LLC entry of node 0 is then invalidated thus a write-back to main memory follows. Finally, the four write operations are ready to be performed by writing and updating the data into the LLC of node 1.

- Subfigure (c): T1 on node 1 and T0 on node 0 are about to perform 2 writes each, in a ping-pong manner (T1-T0-T1-T0). T1's writes are considered as "shared" according to our initial definition while T0's writes are "thread-local". Even though the "shared" writes are fewer compared to case (b), the ping-pong pattern

Table 6.3: Shared Accesses. "Owner" = Last Writer.

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

of (c) is proved much more expensive than (b) since it results in four remote node accesses, four memory write-backs and consequently heavier load on the interconnect.

The expensive case (c) results in counting two shared writes while, (b) counts four. However, by considering the last thread that wrote/updated the object as the "owner" (*last writer*), the case (b) counts one shared write access, while the expensive case (c) counts four shared write accesses. Therefore, it turns out that the "owner" should be the last writer thread because using this heuristic can detect the expensive cases more accuratelly. As a result, this section redefines the classification heuristic for the shared object accesses. Hereafer, the last writer thread of an object is considered as the "owner".

Table 6.3 lists the amount of shared accesses as percentage over total object accesses per parallelism category (excluding "Single-Threaded") considering the *last writer* thread as the "owner". A high amount of shared reads or writes strongly indicates that an application contains shared resources; however, it is not a proof of contention. Three major categories are observed: the "Data Parallel", the "Read & Write Dependencies", and the "Read-Only Dependencies". The following sections discuss each category.

6.3.1 Data Parallel

`Log-regression` and `naive-bayes` fall into the "Data Parallel" category. Having neither considerable shared reads nor shared writes, this category denotes that probably is free of data dependencies. The absence of shared data 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.

6.3.2 Read & Write Dependencies

`Avrora`, `h2`, `xalan`, `akka-uct`, `reactors`, `db-shootout`, and `philosophers` belong to the category of "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 harms locality in case those threads are naively scheduled on different NUMA nodes. Such a phenomenon, inevitably increase the pressure on LLC/Memory, and the interconnect traffic. Consequently, this kind of data dependencies is very likely to prevent those applications from scaling.

6.3.3 Read-Only Dependencies

`Lusearch`, `pmd`, `sunflow`, `als`, `chi-square`, `gauss-mix`, `movie-lens`, `fj-kmeans`, `future-genetic`, `par-mnemonics`, `rx-scrabble`, `scrabble`, `neo4j-analytics`, and `scala-stm-bench7` are placed into the category of "Read-Only Dependencies". A considerable amount of shared reads with negligible or even zero shared writes indicates the existence of shared data and/or data structures. However, such a fact might not lead to increased pressure on the LLC/memory and the interconnect; thus, might not essentially prevent scalability. The F state of the MESIF cache coherency protocol is an optimization to mitigate contention for shared read data (see Section 2.2.1). It essentially allows a clean/unmodified cache line to be shared for reads among the nodes by providing a "copy" to the requesting node avoiding accesses to main memory [GH09]. However, write accesses performed by object-owner threads can still invalidate cached data and retain contention. More specifically, in case the object-owner thread writes an object that is read by other threads, the cache line of the reader is invalidated. Moreover, the updated value is cached in the node of the writer,

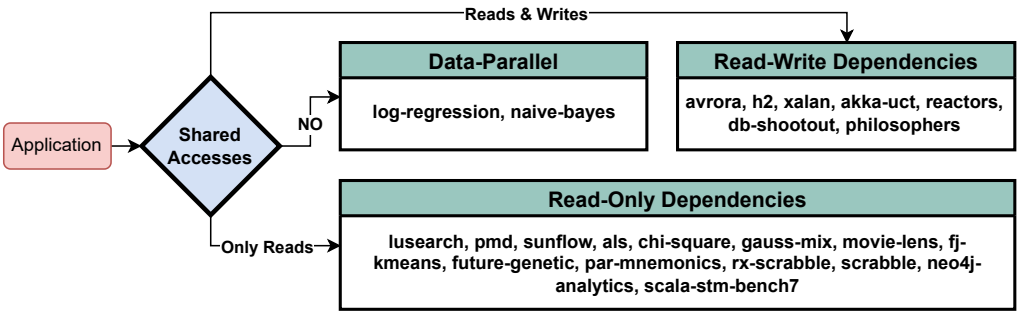


Figure 6.3: NUMA Scalability Characterization: Data Dependencies.

potentially by overwriting other cache lines, thereby leading to more capacity cache misses. We name the above situation as a "write conflict" hereafter.

6.3.4 Summary of Data Dependencies Characterization

Figure 6.3 summarizes the aforementioned examined criteria towards characterizing the assessed applications regarding data dependencies. Three discrete classes of applications seem to exist: "Data Parallel", "Read & Write Dependencies" and "Read-Only Dependencies". Strong dependencies borne by the working data of an application with "Read & Write Dependencies" might cause cache invalidations, remote node accesses and interconnect congestion. All the above undermine the scalability of the application on a NUMA system. On the contrary, "Data Parallel" applications do not have data-dependencies and subsequently, have no such limitations. Finally, an application with "Read-Only Dependencies" might behave either like "Data Parallel" or like having "Read & Write Dependencies". This depends on the extent of the "write-conflicts" effect. The ambiguity borne by the above findings leads to the inevitable correlation between data dependencies and data locality. As a result, the last characterization step focuses on data locality and aims to merge all previous steps.

6.4 Locality

The degree of temporal/spatial regularity in memory accesses is the major factor of interest in non-NUMA architectures because only irregular memory accesses or capacity misses can harm data locality and consequently performance. However, in a NUMA system, the LLC is shared across the nodes. Consequently, the data dependencies might dramatically hurt locality due to "write-conflicts"

Table 6.4: Single vs Dual Node Run 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)	100	100

(see Section 6.3). As a result, the evaluation of data locality with regards to both regularity of memory access patterns and data dependencies is of outmost importance. Such an assessment, is a challenging task due to the multiple aspects and factors involved, especially in the context of NUMA architectures.

6.4.1 Methodology

A direct approach would require to monitor the accessed memory addresses per thread in order to create a detailed map of memory regions that each thread access. This approach can provide a detailed profile of memory access patterns for each thread in order to evaluate the inter-thread data dependencies and the degree of regularity/irregularity of the memory access patterns. However, such an approach would lead to lengthy traces able to provide useful information only during a heavy post-execution phase; thus being incompatible with a flexible runtime approach. Therefore, an empirical methodology is employed as a means to retain flexibility. The assessment of data locality for an application is performed by observing the difference in LLC miss rate between a NUMA and a non-NUMA configuration. The "Single Node" run configuration (see Section 5.2) is used as the non-NUMA configuration while, the "Dual Node" configuration is introduced for NUMA. The characteristics of the Single Node and Dual Node configurations are highlighted in Table 6.4. As can be seen, the Dual Node configuration deploys two NUMA nodes but it has the same number of cores with Single Node configuration. The utilization of equal number of cores by both configurations simplifies the observation of memory behavior and data locality by excluding the potential scalability effects. Therefore, the comparison between the Dual Node configuration and the Single Node configuration isolates the effect of NUMA on the memory behavior of the application. More specifically, the aim of this approach is to reveal how the memory behavior of an application is affected by

Table 6.5: LLC Miss Rate comparison per Parallelism and Data Dependencies Category.

	TLP Bound	R		W		**Embarrasingly Imbalanced**	R		W	
		S	D	S	D		S	D	S	D
RD	als	6	19	44	50	par-mnemonics	8	9	55	55
	chi-square	10	13	60	59	rx-scrabble	4	7	59	61
	gauss-mix	3	3	60	60					
	movie-lens	4	9	51	54					
	neo4j-analytics	24	23	63	61					
DP	log-regression	54	56	57	57					

	Imbalanced	R		W		**Explicitly Parallel**	R		W	
		S	D	S	D		S	D	S	D
RD	pmd	2	2	58	56	lusearch	0	2	54	56
	fj-kmeans	70	71	44	63	sunflow	2	2	64	62
	future-genetic	0	14	48	58	scrabble	2	22	33	72
	stm-bench7	16	31	68	63					
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						naive-bayes	17	22	59	58

the remote LLC and memory. Moreover, the prior characterization steps (Sections 6.2 and 6.3) provide a stable ground for the characterization of data locality by eliminating other affecting factors. Although this comparison methodology is currently applied in a post-execution phase, it has the potential to be applied online in a low overhead manner by the JVM runtime.

The practices that are described in Section 5.2 regarding the experimental process and the extraction of the results (i.e., the amount of run iterations, the results are the average of the 10 last iterations, etc.) are also applied for the Dual Node run configuration.

Table 6.5 presents the Read (R) and Write (W) LLC Miss Rate (%) per application in the Single (S) and Dual Node (D) configurations. Its objective is to highlight the effect of NUMA on data locality by observing the difference in the LLC Miss Rate between the Single Node and Dual Node configurations. The layout of Table 6.5 groups the applications in the categories introduced by Sections 6.2 and 6.3. The horizontal axis hosts the four categories of Parallelism ("TLP-Bound",

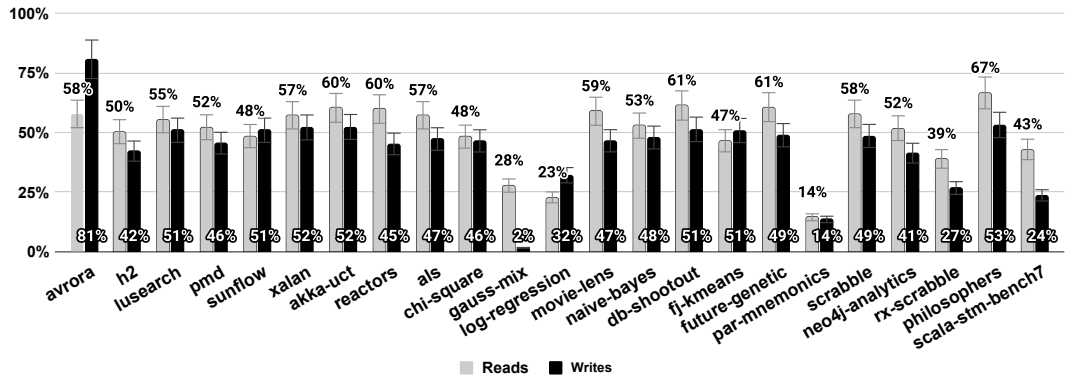


Figure 6.4: Percentage of Read & Write LLC Misses that are served by Remote Node Memory.

"Embarrassingly Imbalanced", "Imbalanced", "Explicitly Parallel"). Note that the "Single-Threaded" applications are excluded. The vertical axis hosts the three categories of Data Dependencies ("Data Parallel" - DP, "Read & Write Dependencies" - RWD, "Read-Only Dependencies" - RD). Empty spaces denote that none of the studied applications belongs to this category. In addition, Figure 6.4 illustrates the percentage of Read & Write LLC misses that are served by remote node memory for each application[2]. Such a metric can potentially be indicative whether the observed difference in the LLC Miss rate is related to "write-conflicts". The following paragraphs discuss the observations regarding data locality in accordance with the already discussed categories in order to place the observations that are related to data locality in the appropriate context.

6.4.2 TLP-Bound

The data locality of "TLP-Bound" applications is negligibly affected by NUMA. Such an observation is expected since the "TLP-Bound" applications usually deploy very few workers, consequently the Linux kerner 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 counter-intuitive cases exist. For example, the `als` and `movie-lens` with a considerable increase (+13%, +5% accordingly) in the observed LLC read miss rate and `neo4j-analytics` with 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 6.1). As a result, this

[2]Memory from local or remote node is accessed upon the occurrence of an LLC miss

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 6.3) imply that the spread threads communicate and share data. However, those applications differentiate regarding the data locality from Single to Dual Node. The fact that the many remote memory reads are acompanied 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, writen/updated by the owner thread and inevitably lead to more LLC misses. On the other 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, it is notable that the `als` and `movie-lens` show the greatest performance degradation among all "TLP-Bound" (see Figure 1.1). `Als`, especially, is a notable case because it is a compute-bound application (see Section 5.4.2) which turns out to be memory-bound when run in a NUMA system due to the data dependencies that harm locality.

6.4.3 Embarrassingly Imbalanced

The negligible difference in the LLC Miss Rate and the number of memory accesses to the remote node reveal the main characteristics of the applications in this category. The multiple workers that "Embarrassingly Imbalanced" applications spawn are not concurrently active, hence those applications behave similarly to the "Single-Threaded" applications. 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 an "Embarrassingly Imbalanced" application even though it is not harmful, it is not likely to be beneficial.

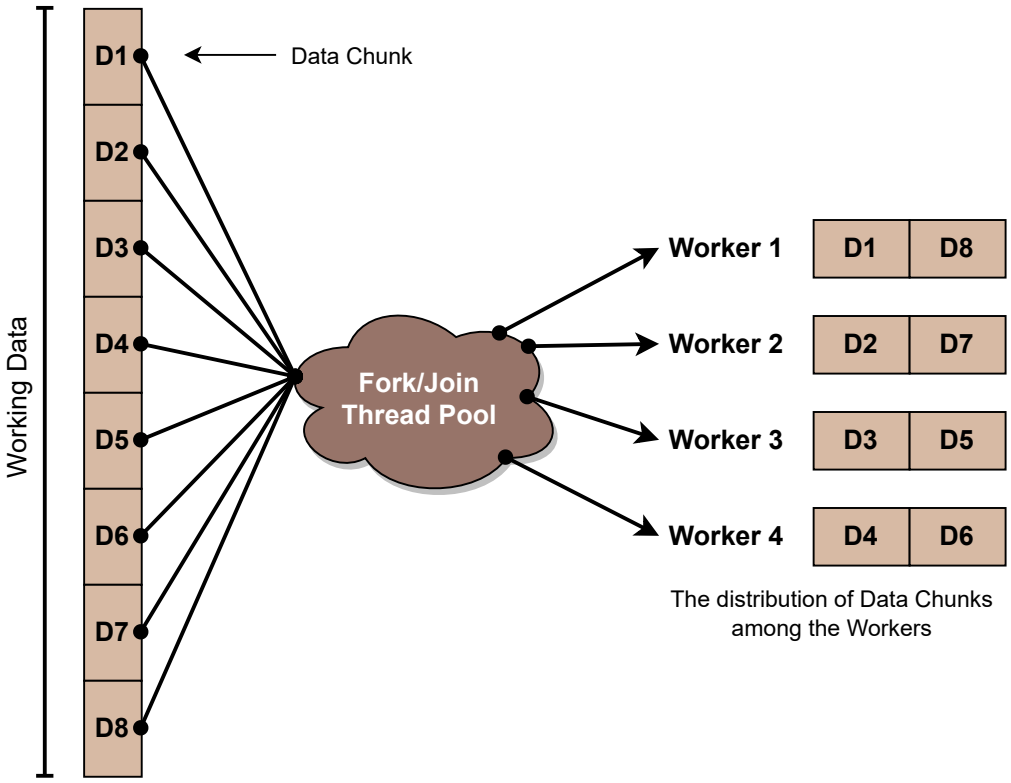


Figure 6.5: Memory pattern irregularity of the fj-kmeans application.

6.4.4 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 in 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, it is notable that the LLC Read Miss Rate of `fj-kmeans` is already high (70%) even in the Single 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 perfrom the centroid calculations. As Figure 6.5 illustrates, the data chunks are distributed to workers on a random and upredictable order. Therefore, each worker processes non "neighbouring" 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 this update phase due to the fact 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 applications in the "Imbalanced" category 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 show 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 dependencies of data breaks the locality by invalidating the already cached data, and consequently leads to more LLC Misses.

The `akka-uct` seems to be affected mostly by write-conflicts over shared read data (similarly to `future-genetic` and `scala-stm-bench7`).

Finally, a notable case is the `reactors` application. It has the highest percentage of shared write object accesses (5.3%, see Table 6.3) 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 transfered in the LLC of the node of the writer, and the cached data (if any) of any other node is invalidated (see Figure 6.2). Therefore, this situation will lead in additional remote node accesses upon the occurance 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`.

6.4.5 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". `Philosophers` is a notable case because the LLC Miss Rates 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 dining philosophers algorithm it implements is a well known synchronization problem over shared resources [TW06]. 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, it has one of the highest BPU Miss Rate (see Section 5.4.2). 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 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%)[3]. 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 1.1.

`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.

6.4.6 Summary of Data Locality Characterization

`Fj-kmeans` and `scrabble` have properties that inevitably bound the locality of data in NUMA due to either direct or indirect irregular memory access patterns. Even though the NUMA configuration impacts the data locality of many applications, this is not necessarily attributed to locality properties of the application itself. Such examples are the `avrora`, `reactors`, and `philosophers` in which the data locality is damaged in Dual Node due to the data dependencies. Consequently, the data dependencies should be also considered as a rather critical factor towards preserving locality while trying to scale on a NUMA system.

6.5 Conclusion of the Scalability Characterization

This chapter evaluated the NUMA scalability properties of the Dacapo & Renaissance applications and revealed several categories that a managed application can belong to. Towards that goal, several high and low level metrics were utilized, such as the high-level imbalance of object accesses, and the amount of shared object accesses as well as, the imbalance in hardware instructions, the LLC Miss Rate, etc.

[3]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.

Section 6.2 studied several metrics relative to parallelism and balance and concluded that five discrete application classes seem to exist: the "Single-Threaded", the "TLP-Bound", the "Embarrassingly Imbalanced", the "Imbalanced", and the "Explicitly Parallel". Section 6.3 revealed that three discrete classes of applications exist: "Data Parallel", "Read & Write Dependencies" and "Read-Only Dependencies" by examining the amount of shared object accesses for each application. Finally, Section 6.4 examined the LLC Miss Rate on the Single and Dual Node configurations in order to assess the data locality of the applications in a NUMA system.

To summarize the characterization of the Dacapo and Renaissance applications, the Table 6.6 incorporates all the above categories. In addition, in order to cross-validate the findings of this chapter the perfromance of the Dacapo and Renaissance application is measured when those applications are free to scale on both NUMA nodes. The applications were run on 2 NUMA Nodes with 16 CPU Cores. This run configuration is refered as "All Nodes" hereafter. The measured performance of "All Nodes" configuration is compared against the baseline "Single Node" configuration (1 Node, 8 Cores). Moreover, all applications were run with 16 threads (wherever possible) in both configurations. The measured performance of each application is presented in Table 6.6. The most scalable applications are among the "Explicitly Parallel" applications without "Read-Only Dependencies" or "Data Parallel". This observation is expected because most of those applications are not bound by data dependencies that damage the locality of data. Consequently, the application threads can run on any node and exploit scale-out resources without drawbacks. Exception is the `scrabble` application which contains irregular memory access patterns as explained in Section 6.4. This case along with the `fj-kmeans`, suggest that applications with irregural memory access patterns cannot take advantage of a NUMA system even though they exhibit parallelism and balance.

Another observation is that none of the "Imbalanced" applications is able to scale. This fact validates the impact of data dependencies on data locality and the impact of low parallelism/balance on NUMA scalability.

The observed performance of "TLP-Bound" and Embarrassingly "Imbalanced" applications suggests that *typically* those applications are incapable of taking advantage of the scale-out resources due to the lack of thread-level parallelism. Therefore, scaling those applications in multiple NUMA nodes should be avoided in order to avoid performance losses. Nevertheless, it should be noted that, an application with limited thread-level parallelism can still benefit from a NUMA system

Table 6.6: NUMA Characterization of Dacapo & Renaissance Applications.

	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	

for memory-related reasons (i.e., due to larger LLC/memory capacity). Unfortunately, none such case is included in the application set that is studied by the current thesis.

Chapter 7

NUMA Optimizations for Managed Workloads

7.1 Introduction

The previous chapters have presented new tools and they have augmented the understanding of readers of Dacapo and Renaissance applications. More specifically, Chapter 4 introduced two new tools which can both obtain useful hardware and application metrics at runtime. In addition, Chapter 5 studied the memory behavior of the deployed applications (i.e., Dacapo, Renaissance), while Chapter 6 characterized those applications as per their NUMA scalability by examining several related properties. On that ground, this chapter aims to address one final question: "To what extent the aforementioned research findings and tooling arsenal can be practically utilized towards improving the performance of a managed application on a NUMA system?". This chapter aims to exploit those findings as a guide to prototype novel, application-agnostic techniques for the JVM towards improving the performance of a managed application on a NUMA machine.

7.2 Overview

Chapter 6 has classified the managed applications into groups as per their NUMA scalability properties. It has not only concluded to several application categories, but also it cross-validated those categories by measuring the performance of each application when running on a NUMA system (Table 6.6). This table essentially validates that the applications within the same category are homogeneous because

most of the applications follow the same trends. Leveraging this knowledge, the problem of poor scalability (or even performance degradation) on a NUMA system can be approached per category rather than per application. Therefore, an optimization approach for each group can be designed that would increase performance on a NUMA system (if applied) or at least avoid the presence of existing overheads when running with unoptimized/naive conditions (see Chapter 1). The classification of Chapter 6 provides awareness whether a NUMA system can benefit a managed application or not, in a beforehand manner. Considering the scenario of having a managed application which should be executed on a NUMA machine, we can conclude to the following guidelines, based on the findings of Table 6.6:

- **(G1)** The application is very likely to be penalized by the NUMA system in case the former belongs to the "Single-Threaded", "TLP-Bound", or "Embarrassingly Imbalanced". As a result, the application should be bound to only one node. ("Single Node" run configuration, see Section 5.2).

- **(G2)** On the other hand, in case the application is either "Imbalanced" or "Explicitly Parallel", it is uncertain whether it is worthwhile to let it scale ("All Nodes" run configuration, see Section 6.5), or not. For example, even an "Explicitly Parallel" application might lose performance in the "All Nodes" configuration, in case it bears irregular memory access patterns (like `scrabble`) or malicious data dependencies (like `db-shootout` and `philosophers`). In addition, the same applies for the "Imbalanced" applications like `fj-kmeans` which has irregular memory access patterns, and `avrora`, `akka-uct`, and `reactors` which have restrictive data dependencies. Consequently, in case an application is either "Imbalanced" or "Explicitly Parallel" it is not totally clear whether it should scale, or not. Hence, an effective distinction between the different cases is needed.

Considering the above guidelines it is clear that optimizing performance in an application-agnostic manner is a challenging task. The "grey areas" of the "Imbalanced" and "Explicitly Parallel" applications inevitably require an online analysis of the running application in order to conclude whether the "Single Node" or the "All Nodes" run configurations should be used. However, the more detailed profiling is deployed, the more overhead is going to be paid. For example, if all the NUMA scalability properties (Parallelism, Data Dependencies and Locality, see Chapter 6) are taken under consideration, an ultra-high profiling overhead price

should be paid mostly due to NUMAProfiler (see Section 4.3.10).

An alternative less overhead-prone approach would be: to deploy only PerfUtil in exchange for a low-overhead, but less detailed profiling *and* cover the missing profiling information by spending some time in running the application in both configurations in order to observe its performance. Such a "trial-and-error" approach can enable a decision-making mechanism that is driven by real-time observations instead of predictions. Therefore, it can conclude to reasonable decisions and avoid the extensive penalization of the application due to profiling.

In this chapter, the above guidelines are utilized as a roadmap towards implementing a decision-making component in MaxineVM in order to dynamically apply the appropriate run configuration in an application-agnostic manner. Moreover, the primary priority of such an approach should inevitably be the low overhead. As a result, two major conditions arise as the key-prerequisites:

- **(P1)** Dynamic decision-making at runtime with low overhead.
- **(P2)** Profiling data available at runtime with low overhead.

The deployment of PerfUtil is rather a one-way for **P2** due to the low overhead utilization of the Hardware Performance Counters it offers. **P1** is up to the creativity of the JVM developer.

To summarize, we have discussed the new opportunities that derive as a product of the earlier research findings of this thesis. It is clear that the Memory Characterization (Chapter 5) and the NUMA Scalability Characterization (Chapter 6) studies have created the conditions for novel dynamic approaches towards improving performance of a managed application on a NUMA system in an application-agnostic manner. In addition, we have briefly highlighted the key-prerequisites to seize those opportunities and the role of PerfUtil, one of the two new tools for MaxineVM introduced by this thesis.

The following sections present the design and implementation of an application-agnostic dynamic decision-making mechanism at run-time that periodically monitors the execution and applies the most propriate run configuration towards improving performance of a managed application on a NUMA machine. The **P1** and **P2** key-prerequisites are further discussed along with the design and implementation challenges they imply. Finally, the introduced overhead and the experimental results of the mechanism are assessed and presented.

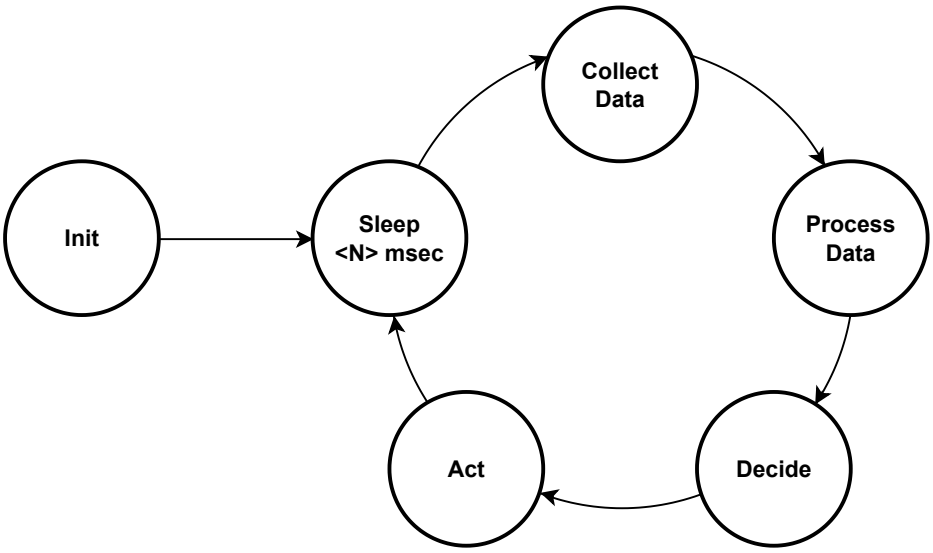


Figure 7.1: Overview of the Awareness Thread actions.

7.3 Mechanism

A background thread monitoring design was selected in order to meet the requirements of **P2**. More specifically, a new VM-internal thread runs in parallel with the application threads (in the background). The new thread should be a daemon that is dedicated to, and dynamically coordinates, the process of profiling and decision-making. The experience obtained from building new tools in the scope of this thesis has taught us that excessive overhead is introduced in case additional duties are assigned to the application threads (see Section 4.3.10). As a result, the choice of a new dedicated daemon thread aims to overcome this bottleneck by isolating the application threads from distractive functionalities and consequently preserve performance with minimal obstructions. Nevertheless, the obstructions are inevitable due to the extra context switches that are expected to be paid by deploying an additional thread because the computing resources are limited. However, this price is expected to be negligible, especially in comparison to the expected performance gains.

The core component of the optimization mechanism is the "Awareness Thread". The actions of the Awareness Thread are briefly illustrated in Figure 7.1. The goal of the mechanism is to successfully detect in which category ("Single-Threaded", "TLP-Bound", "Embarrassingly Imbalanced", "Imbalanced", "Explicitly Parallel") a running application belongs to, and to apply the corresponding configuration afterwards. It is initialized along with the rest VM-internal threads (`main`, `VmOperations` etc.) during the `STARTING` phase of MaxineVM. After initialization, the Awareness Thread

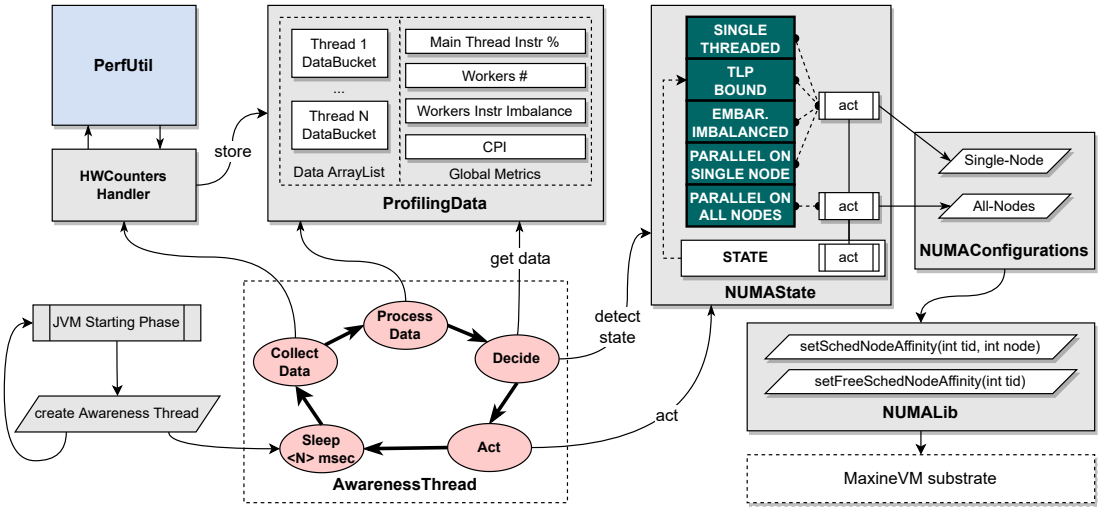


Figure 7.2: Overview of the Optimization Mechanism.

becomes a daemon, iterates over a set of actions that are depicted in Figure 7.1 (Sleep, Collect Data, Process Data, Decide, Act), periodically assesses the running application state, and acts accordingly in order to achieve the best possible performance. The optimization mechanism can be enabled with the `-XX:+NUMAOpts` MaxineVM option. The optimization mechanism is illustrated in Figure 7.2. The following sections present each action of the Awareness Thread along with the corresponding part of the mechanism.

7.3.1 Sleep

Any iteration of the Awareness Thread starts after a sleep time interval. The sleep period length is timed in milliseconds and controlled by the user with the `-XX:NUMAOptInterval` MaxineVM option. This time interval essentially regulates the operation frequency of the Awareness Thread and subsequently how often a new decision is taken. The time interval is set to 200ms after experimentation.

7.3.2 Collect Data

As explained earlier, we aim to exploit the profiling capabilities of MaxineVM while keeping the overhead low. More specifically, it is possible to meet the requirements of **P2** by utilizing PerfUtil because it comes with very low overhead. However, a trade-off between simplicity and potential capabilities arise. Note that the goal of the current work is to showcase the potential practical opportunities towards improving

performance on a NUMA machine by prototyping a dynamic mechanism into the JVM runtime. To that extent, we monitor the retired hardware instructions and CPU elapsed cycles per thread in order to obtain an approximate but clear image of the workload and performance. MaxineVM has been modified so that each new thread instance notifies PerfUtil to enable the corresponding Hardware Performance Counters for itself during its initialization. During the "Collect Data" step the Awareness Thread calls the `HWCountersHandler` to collect the monitored events' values for all active application threads. The `HWCountersHandler` stores the collected values in the `ProfilingData` buffer in order to be available for the next step of processing. The buffer organizes the data per thread as `DataBucket` objects and stores them in an `ArrayList`. A `DataBucket` object stores the following fields: *threadId*, *tid*, *threadName*, *count of retired hw instructions*, *percentage of hw instructions over total*, *elapsed CPU cycles* and *CPI*.

7.3.3 Process Data

The purpose of this action is to process the collected data in order to calculate all the required key-metrics to support the decision-making step.

The key-metrics are:

1. the percentage of main thread instructions over total retired instructions,
2. the number of worker threads,
3. the worker threads instructions imbalance, and
4. the Cycles per Instruction (CPI).

Based on empirical observations an application thread can be considered as a worker in case it retires more than 4% of total hardware instructions. Using this number all worker threads are succesfully detected along with negligible false-positives (auxiliary threads that are mistakenly considered as workers). However, the false positives last only for a short period of time and only in applications with very few threads, hence they negligibly affect the mechanism.

7.3.4 Decide

The decision-making algorithm is abstractly highlighted in Algorithm 1. The goal of this action is to decide in which state the application currently is. The state is stored in

Algorithm 1 Decision-making algorithm

```
if mainThreadHWInstructionsPercentage > 80 then
    state ← SINGLE THREADED
else if numOfWorkers ≤ NUM_OF_SINGLE_NODE_CORES then
    state ← TLP_BOUND
else if workerInstructionsImbalance > 90 then
    state ← EMBARRASSINGLY_IMBALANCED
else
    if singleNodeCPI ≡ 0 then                                  ▷ parallel for the first time
        state ← PARALLEL_ON_SINGLE_NODE                        ▷ run on single node
    else
        cpiMargin ← singleNodeCPI · marginFactor
        if (allNodeCPI ≡ 0) ∨ (allNodeCPI ≤ (singleNodeCPI + cpiMargin)) then
            state ← PARALLEL_ON_ALL_NODES                      ▷ run on all nodes
        else
            state ← PARALLEL_ON_SINGLE_NODE                    ▷ back to single node
        end if
    end if
end if
```

the `state` variable of `NUMAState`. The `state` variable is an enum of type `STATE` and its possible values are: `SINGLE_THREADED`, `TLP_BOUND`, `EMBARRASSINGLY_IMBALANCED`, `PARALLEL_ON_SINGLE_NODE`, and `PARALLEL_ON_ALL_NODES`. The first three conditions check whether the application fits into one of the "Single-threaded", "TLP-Bound", or "Embarrassingly Imbalanced" categories. An application is considered as "Single-Threaded" if the main thread significantly dominates the retired hardware instructions (>80%). If the application is not "Single-Threaded", but it deploys equal or fewer workers than the number of available cores it is considered as "TLP-Bound". The degree of imbalance in hardware instructions equal to 90% of the worker threads has been found as adequate to successfully detect the "Embarrassingly Imbalanced" category. Note that, the specific values of the metrics that are utilized for detecting the "Single-Threaded" and "Embarrassingly Imbalanced" applications (main thread retired hardware instructions percentage = 80%, and imbalance in hardware instructions of worker threads = 90% respectively) were determined based on experimentation with the profiling data.

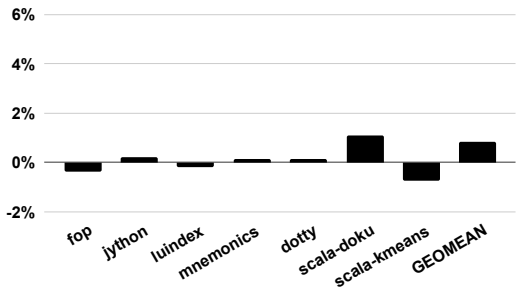
In case an application has not fallen into one of the aforementioned categories, it is considered as "Parallel" ("Imbalanced" or "Explicitly Parallel") and a conservative strategy is followed afterwards. As explained in **G2**, the decision is critical at this point because it can affect performance either positively or even negatively. The

mechanism evaluates the performance of the application in two transitional steps and then it decides if it should let the application scale or not. As a first step, the application is forced to remain on a single NUMA node (`PARALLEL_ON_SINGLE_NODE` state) for one time interval. At the next interval the application is forced to run on all NUMA nodes (`PARALLEL_ON_ALL_NODES` state). After those two time intervals, the mechanism has measured the `singleNodeCPI` and the `allNodeCPI`. The frequency of the system is fixed (see Section 5.2), 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 allowed to continue running on all NUMA nodes if the `allNodeCPI` is lower (better) or equal to the `singleNodeCPI` + `cpiMargin`. Otherwise, the application should settle back to single node.

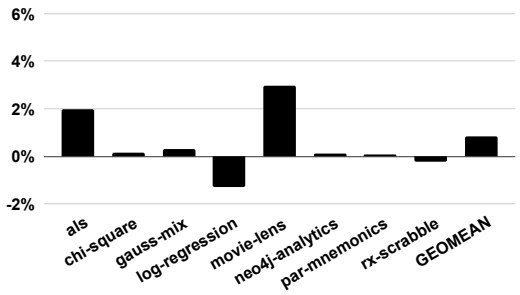
The `cpiMargin` is a synthetic value that makes the decision-making mechanism more tolerant regarding false-negative decisions. The empirical observations have revealed that a strict comparison between `singleNodeCPI` and `allNodeCPI` often results in false-negative decisions (return to single node while it should be scaling). These wrong decisions were significantly mitigated by adding the `cpiMargin` to the `singleNodeCPI` that essentially increases the latter by N%. To find the value of N, we compared the CPI of all "Parallel" applications in "Single Node" (1 NUMA node, 8 cores) and "All Nodes" (2 NUMA nodes, 16 cores) configurations. We observed that the scalable applications had at least 15% lower (better) `allNodeCPI` than `singleNodeCPI`; thus, we define N = 15%.

7.3.5 Act

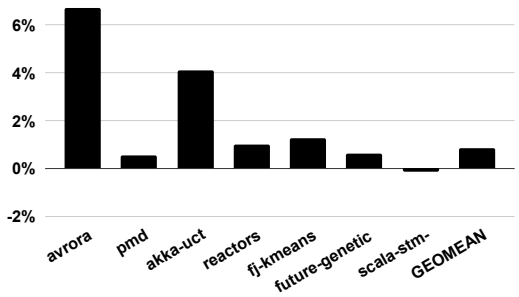
This is the last action of the Awareness Thread cycle before it sleeps again. The act step succeeds the decision-making, consequently the mechanism is already aware of the current state of the application. The `STATE` enum contains an abstract `act()` method; hence each enum value should implement its own `act()` method. The Awareness Thread calls the `act()` method of the `state` variable and consequently the corresponding action is taken through the `NUMAConfigurations`. The latter interfaces with the native system calls of Linux libnuma via the `NUMALib` utility of MaxineVM and the substrate. As illustrated in Figure 7.2 the `act()` of `PARALLEL_ON_ALL_NODES` state leads to the "All-Nodes" configuration while the rest 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.



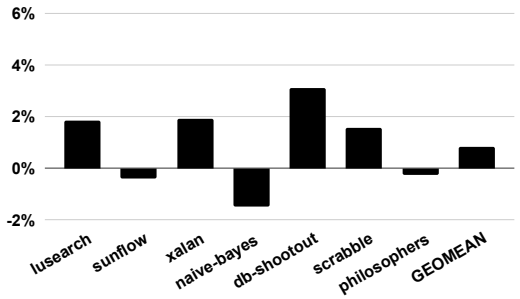
(a) Single-Threaded



(b) TLP-Bound & Embarrassingly Imbalanced



(c) Imbalanced



(d) Explicitly Parallel

Figure 7.3: Mechanism Overhead

7.4 Optimization Mechanism Overhead

Prior to evaluating the impact of the mechanism on performance, an overhead assessment is required. We use an unmodified build of MaxineVM as the baseline (`MaxineVM_vanilla`) which is explicitly pinned (with taskset) to the Single Node. We compare the `MaxineVM_vanilla` against `MaxineVM_fakeOpts`, another MaxineVM build which is equipped with a modified version of the proposed mechanism. The `MaxineVM_fakeOpts` build aims to isolate the mechanism functionalities and effectively measure their overhead by excluding the effect of NUMA scaling. More specifically, the `PARALLEL_ON_ALL_NODES` state can be reached as usual, however it points to the "Single Node" configuration. This way, the mechanism is fully functional while at the same time the execution time is not affected by NUMA-related factors that cause performance variations. In addition, the applications were run with 16 threads, wherever possible, in both configurations. The average execution time from five runs of the `MaxineVM_fakeOpts` is compared against the average execution time from five runs of `MaxineVM_vanilla`. The results are presented in Figure 7.3. The applications

are grouped per parallelism category in subfigures. The values in Figure 7.3 refer to the average execution time of `MaxineVM_fakeOpts` normalized to the average execution time of `MaxineVM_vanilla`. As can be observed the overhead of the mechanism varies, however it is low (geometric mean = 0.84%). The highest overhead is observed in `avrora` (6.74%) and the lower in `naive-bayes` (-1.49%). Regarding the negative overhead values (in `log-regression`, `rx-scrabble`, `scala-stm-bench7`, `sunflow`, `naive-bayes`, and `philosophers`) the absolute difference of the execution times ranges from 3-92ms. Consequently, the negative values are very low and are attributed to the error margin. Considering that the mechanism is invoked in the same rate for all applications and any NUMA-related effect is excluded, the variation and the negative overhead values are attributed to:

1. the additional context switches caused by the mechanism,
2. the dynamic features of Java (JIT compilation, Garbage Collection), and
3. the background dynamic activity of the system (kernel, services, and other processes).

7.5 Performance Evaluation

To measure the effect of the optimization mechanism, the `MaxineVM_vanilla` in the "All Nodes" configuration (2 NUMA nodes, 16 CPU cores) is considered as the baseline. Consequently, such a comparison highlights the effect of the optimization mechanism over naive performance of each application on a NUMA system. The effect of the optimization mechanism on each application and the geometric mean are presented in Figure 7.4. Each value is the average of five executions. As can be observed from the Figure 7.4, in general, the optimization improves performance by 8% (geometric mean 1.08x). In addition, the optimization benefits some applications (i.e., `scrabble`, `fj-kmeans`, `reactors`, `db-shootout` and more), while others are penalized (i.e., `naive-bayes`, `sunflow`, `scala-stm-bench7`).

The most benefited applications by the optimization mechanism (i.e., `scrabble`, `fj-kmeans`, etc.) are those which are heavily penalized by a NUMA system when running on a NUMA-unaware manner (see Table 6.6). Therefore, the results confirm that the optimization mechanism is able to detect such a performance degradation. This is achieved by initially pinning the application on one NUMA node and afterwards letting the application to scale but only for a limited amount of time (optimization

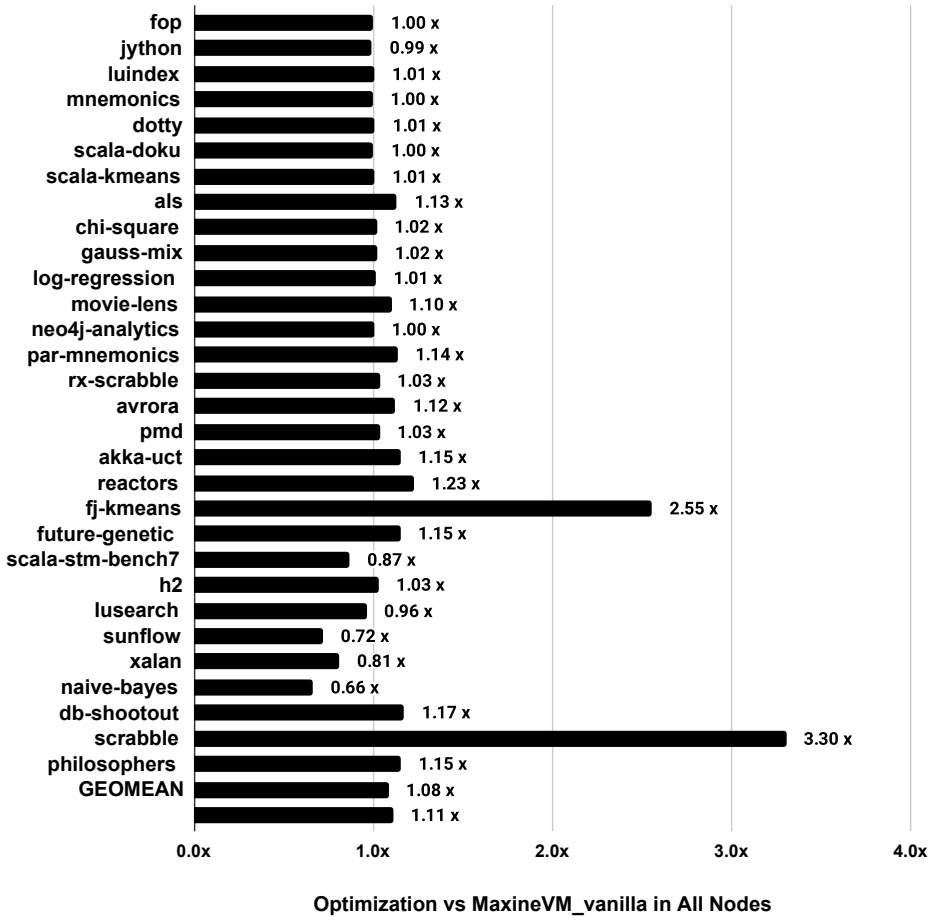


Figure 7.4: Performance evaluation of the optimization mechanism in comparison to the performance of the `MaxineVM_vanilla` in the "All Nodes" configuration.

interval). During those steps, the optimization mechanism monitors the CPI, and therefore, it decides which configuration should be applied for the remaining of the running application. Consequently, the larger the penalization of NUMA, the most accurate the optimization mechanism is, with regard to those cases.

On the contrary, most of the applications that are penalized by the optimization mechanism (i.e., `naive-bayes`, `sunflow`, etc.) are those which are able to effectively scale on a NUMA system when running on a NUMA-unaware manner (see Table 6.6). This observation is not only counter-intuitive, but it also reveals the existance of one inefficiency that is discussed in the following section.

7.5.1 Cold Effect

The optimization mechanism is expected to achieve similar - to Table 6.6 - speedups for the scalable applications (i.e., for `naive-bayes` that shows 1.86x speedup in Table 6.6). However, this is not the case. As explained before, the initial configuration of the optimization mechanism is to pin the application in one NUMA node. This strategy, inevitably leaves the other NUMA node unutilized. Therefore, the caches of the unutilized NUMA node will be "cold" when the optimization mechanism lets the application to run on both NUMA nodes. As a result, this effect (is refered as "cold effect" hereafter) might lead the optimization mechanism to a false-negative decision. In that occasion, the decision of utilizing all NUMA nodes which benefits a scalable application will be delayed (for at least one decision interval) or even never taken. This inefficient situation inevitably penalizes the execution time because the scalable application utilizes only one NUMA node instead of two.

In addition, the cold effect undermines performance even if the mechanism avoids the false-negative decision and finally utilizes both NUMA nodes. The application threads that migrate to the cold NUMA node will be facing remote node accesses and even cache misses because their working data is already cached in the initial NUMA node, thereby increasing the "All Node" CPI. Consequently, the mechanism might be influenced (at the next decision interval) to move again the application threads back to one NUMA node. Even though, the migration cost is a price that needs to be paid in exchange for long-term performance improvement, it turns out to a bottleneck if it is paid in an inefficient and often manner.

To that end, it is clear that the optimization mechanism comprises the cost of migration as an additional overhead quantity due to the cold effect. Unfortunately, this quantity cannot be measured in a direct manner because it depends on several dynamic and low-level factors such as the state of the caches. However, it can be approached by counting the migrations caused by the mechanism, correlate them with performance observations, and potentially mitigate their cost by avoiding those unecessary.

7.5.2 Optimization-S

Motivated by the revealing of the cold effect, we measure the migrations caused by the optimization mechanism with the aim of correlating them with performance. The two following cases are considered as migrations:

1. "$SingleNode$" $\rightarrow$ "$AllNodes$"

Table 7.1: Default Opt. Mechanism

Application	# of Migrations
avrora	35
pmd	0
akka-uct	192
reactors	0
fj-kmeans	3
future-genetic	718
scala-stm-bench7	71
h2	115
lusearch	30
sunflow	31
xalan	28
naive-bayes	42
db-shootout	177
scrabble	62
philosophers	41

Table 7.2: Modified Opt. Mechanism

Application	# of Migrations
avrora	32
pmd	0
akka-uct	122
reactors	0
fj-kmeans	2
future-genetic	689
scala-stm-bench7	72
h2	24
lusearch	22
sunflow	7
xalan	3
naive-bayes	1
db-shootout	21
scrabble	3
philosophers	3

2. "*AllNodes*" → "*SingleNode*"

Table 7.1 presents the number of migrations for all "Parallel" applications. The numbers refer to all migrations happened across the execution of each application (end-to-end) and are the average of five executions. To effectively assess the effect of migrations, we slightly modify the optimization mechanism as follows. As long as the mechanism takes the same decision for a "Parallel" application, the operation frequency of the mechanism is gradually reduced. More specifically, if the current decision is the same with the previous one, the optimization interval is increased by 200ms. This modification aims to "stabilize" the decision mechanism in order to avoid unneccessary interruptions and ineffective migrations. Essentially, it is based on the heuristic that a repetitive decision is less likely to be a coincidence. In addition, in case the decision of the mechanism alters at any point, the optimization interval is reset to its initial value (200ms). As a result, the mechanism is still able to adapt in case the established decision is no longer beneficial. The number of migrations caused by the modified optimization mechanism are presented in Table 7.2. In addition, Figure 7.5 evaluates the performance of the modified optimization mechanism (OptS) against the `MaxineVM_vanilla` in the "All Nodes" configuration (2 NUMA nodes, 16 CPU cores).

By comparing the performance of the penalized applications of Figure 7.4 with the same of Figure 7.5, (`naive-bayes`: 0.66x, `sunflow`: 0.72x, `xalan`: 0.81x, `scala-stm-bench7`: 0.87x, `lusearch`: 0.96x), it can be observed that the

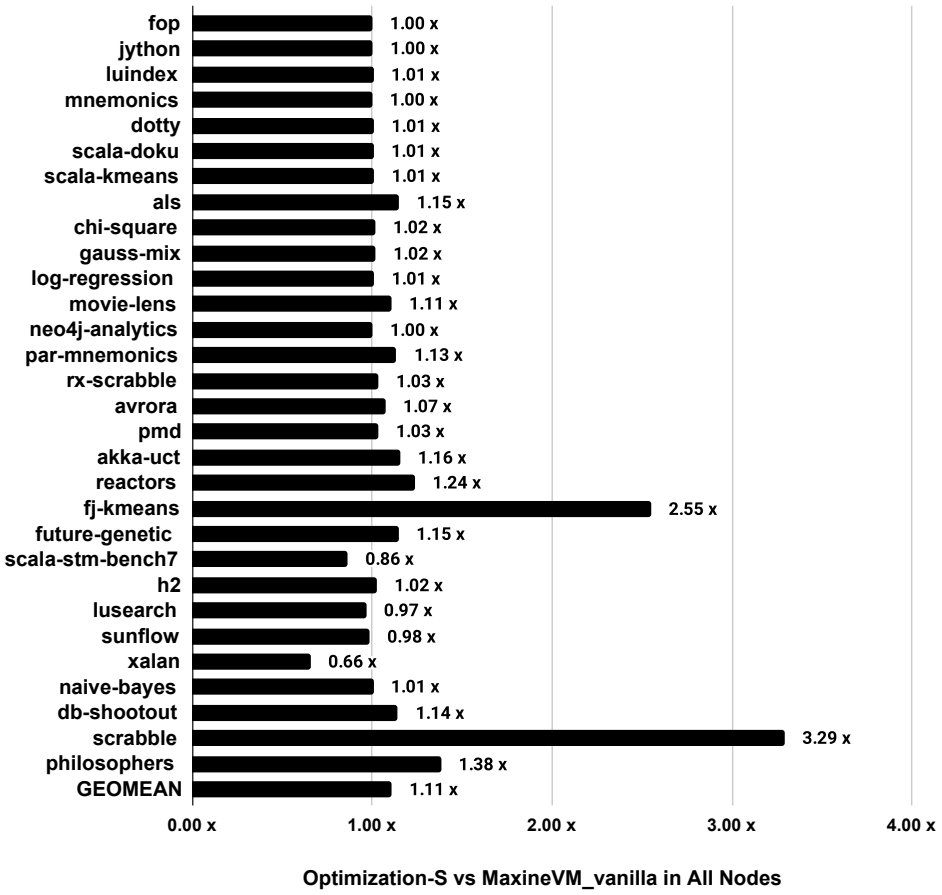


Figure 7.5: Performance evaluation of the modified optimization mechanism in comparison to the in comparison to the performance of the `MaxineVM_vanilla` in the "All Nodes" configuration.

modification benefits the `naive-bayes` (1.01x) and `sunflow` (0.98x), does not affect `lusearch` (0.97x) and `scala-stm-bench7` (0.86x), while it further penalizes `xalan` (0.66x). Moreover, this performance differentiation is reflected to the migrations. More specifically, the modification of the mechanism reduces the migrations for the benefited `naive-bayes` and `sunflow` (from 42 to 1, and from 31 to 7 respectively), while it has no significant impact on the migrations of the unaffected `lusearch` and `scala-stm-bench7`. However, the migrations are also reduced for the penalized `xalan` (from 28 to 3) which is counter-intuitive. These observations lead to the following conclusion. The scalability of `naive-bayes` and `sunflow` is indeed penalized by frequent migrations. By reducing the frequency that the mechanism operates, the migrations are avoided, hence those applications can benefit from the

utilization of two NUMA nodes. Nevertheless, this does not apply to all applications. The example of `xalan` suggests that some applications are not benefited by infrequent decisions, thereby indicating that the behavior of those applications needs a more precise management. 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 $\sim 80\%$ and is multithreaded for the remaining $\sim 20\%$. Consequently, a trade-off between potential scalability benefits and precise decisions arises. Finally, it should be noted that the geometric mean of all applications confirms that the modified optimization outperforms the initial optimization in general (1.11x from 1.08x).

7.6 Summary

This chapter has presented a dynamic, and application-agnostic mechanism that improves the performance of a managed application by 11% on average. This mechanism is implemented into the runtime of MaxineVM, hence it operates during the execution. It assists the VM to decide whether a running application should be pinned on one NUMA node ("Single Node" run configuration) or should expand to run on the two NUMA nodes of the system ("All Nodes" run configuration). The decision is driven by online profiling data that are provided by the PerfUtil with low overhead (geometric mean = 0.84%). The proposed optimization mechanism loops over a set of actions in order to coordinate the collection of the profiling data, the processing of the profiling data, the decision-making algorithm, and the application of the decided run configuration.

The evaluation of performance has revealed several inefficiencies that may be occurring while migrating from one to two NUMA nodes and vice-versa. To tackle those inefficiencies, the mechanism was modified ("Optimization-S") in order to gradually reduce its frequency of operation and avoid unoptimal migrations. Even though the modification improved the overall performance (from 1.08x to 1.11x), it also highlighted that some applications require even more fine-grained management.

Chapter 8

Conclusions and Future Research Directions

The ever-expanding amount of data that is processed by modern software has increased the demand for compute and memory resources. NUMA designs have been introduced to provide sufficient resources by aggregating excessive amount of CPU cores and DRAM. However, the effective scalability of software on NUMA hardware has turned out to a non-trivial task due to the particularities of those designs. On that ground, managed applications struggle even more to scale due to the additional layer of the MRE. As a result, research interest has been shifted towards the study and the efficient exploitation of NUMA hardware by MREs.

This thesis is involved to the above research field by studying the memory behavior, and the scalability bottlenecks of several managed applications. Moreover, it proposes two new tools and a dynamic mechanism for MaxineVM towards improving the performance of a managed application on a NUMA machine.

Section 8.1 summarizes the work conducted in the context of this thesis while Section 8.2 proposes several future research directions that can further evolve the findings of this work.

8.1 Thesis Summary

The work conducted by this thesis is articulated as follows:

Chapter 2 outlines the topics related with the thesis and discusses all the prerequisites aspects to enable the understanding of this thesis. More specifically, it briefly discusses the major concepts of the MREs, explains how a JVM works,

and introduces the reader to MaxineVM and metacircularity. Moreover, it describes the advent of the Non-Uniform Memory Access (NUMA) architectures and how remote memory affects performance. In addition, it introduces the reader to Hardware Performance Counters. Note that, this chapter unfolds various sub-topics within the aforementioned areas, such as metacircularity in the JVM, the MESIF cache coherency protocol and more, which are necessary for the readers' better understanding.

Chapter 3 discusses several challenges in regard to MREs in the context of the NUMA architectures, outlines, and reviews the related research works from the literature.

Chapter 4 briefly presents a novel approach that aims to effectively correlate the application properties with hardware behavior, in order to achieve a better understanding of the behavior of managed applications when deployed on a NUMA system. Essentially, this chapter introduces two new tools for MaxineVM, that compose the research platform that is proposed and utilized by this thesis. It describes in detail the two profiling tools of the proposed research platform and discusses the challenges faced and the key design choices taken. More specifically, the two tools are: the *PerfUtil*, a low-level microarchitectural profiler, and the *NUMAProfiler*, a high-level application-layer profiler. In particular, PerfUtil monitors the hardware performance counters, while NUMAProfiler probes the runtime layer of MaxineVM and monitors object-related metrics.

Chapter 5 presents a study on the memory behavior of the Dacapo and Renaissance applications. Towards that objective, the PerfUtil and NUMAProfiler tools are utilized. The study is presented per benchmark suite in order to provide a suite-wide view to the reader. The study comprises a high-level and a low-level point-of-view profiling. The co-utilization of the above, augment the understanding of the Research Community regarding the popular Dacapo and Renaissance benchmarking suites for managed runtimes. In addition, it provides a solid foundation towards analysing the behavior of those applications in the context of a NUMA architecture.

Chapter 6 presents a NUMA scalability characterization for the Dacapo and Renaissance benchmark suites in order to demystify the necessary and sufficient conditions under which NUMA architecture can be beneficial for an application. This chapter gradually assess several NUMA scalability-related properties, such as parallelism, balance, data dependencies, and data locality for each application. Each property assessment results to a set of homogeneous application groups. The intersection of the groups formalizes a NUMA characterization table where each

application belongs to a single category. This study not only succeeds in characterizing the Dacapo and Renaissance applications, but also sheds light into the existing application categories and their behavior in a NUMA system. The findings of this chapter augment the understanding regarding the properties that are required by a managed application in order to take advantage of NUMA hardware.

Chapter 7 describes and demonstrates a dynamic, and application-agnostic mechanism that improves the performance of a managed application on a NUMA system. This mechanism is implemented into the runtime of MaxineVM and operates online during execution. It assists the VM to decide whether the application should be pinned on one NUMA node or should expand to the two NUMA nodes of the system. The decision is based on online profiling data that are provided by the PerfUtil with low overhead (geomean is 0.84%). The mechanism iterates over a set of actions in order to coordinate the collection of the profiling data, the processing of the profiling data, the decision-making algorithm, and the application of the decided run configuration. It is demonstrated that the performance of a managed application is improved by 11% when this dynamic mechanism operates and assists the execution.

8.2 Future Research Directions

Towards improving the performance of a managed application in a NUMA system, this thesis has resulted in several research findings. The two tools, the memory behavior study, the NUMA scalability study, as well as the dynamic optimization mechanism can be leveraged as an effective foundation for future research. Moreover, it should be noted that even though this work focuses on Java applications and utilizes MaxineVM, it is neither language nor VM-specific. All presented tools, techniques and research approaches are applicable to any VM. In addition, the working principles of the optimization mechanism can be adopted by any VM. More specifically, regarding the JVMs, the implementation of the optimization mechanism is orthogonal and acts complementary to any JIT compiler (i.e., Graal) or GC implementation (i.e., G1). The optimization mechanism is driven by performance observations that derive by the Hardware Performance Counters while it relies on the OS scheduling; hence it does not interfere with the aforementioned performance-critical components of a JVM. The G1 collector and `-XX:+UseNUMA` especially, can seamlesly cooperate with the presented mechanism since the latter will detect the need for spreading the threads across the NUMA nodes and will allow them to take advantage of the fragmented heap spaces of

G1. The following list articulates potential research directions that can derive from the current work:

- **Larger-scale and asymmetric NUMA machines:** This thesis studied the effect of NUMA architecture on managed application, in the simplest form of this architecture (i.e., a two-node NUMA manchine). Such a choice was a result of the ever-increasing complications of NUMA architectures as more nodes are utilized. The resulted complexity of the current work justifies this choice without ignoring any substantial principle of the NUMA architecture. Nonetheless, a wide variety of NUMA systems exists. Consequently, the research methodology and the optimization mechanism presented by this thesis could be applied and be extended for different and potentially larger-scale NUMA systems. Some notable examples of such systems are the NUMAScale systems, and the asymmetric NUMA systems. A NUMAScale system takes advantage of the NumaConnect technology to aggregate multiple server blades to a single system with shared memory and cache coherency [Rus]. Approaching the scalability of managed application on a NUMAScale system additional challenges such as, the even higher memory access latency across the different blades, are introduced. In addition, the so-called "asymmetric" NUMA systems connect the aggregated NUMA nodes with links of different (asymmetric) bandwidth [LQF15]. Therefore, it is clear that additional challenges need to be tackled by an MRE towards exploiting the NUMA scalability of such a system.

- **Towards more efficient and sophisticated profiling techniques:** The profiling of a managed application is not a trivial task. As shown in the case of NUMAProfiler, collecting a high-level profiling from the runtime layer is overhead-prone and can reduce the profitability of the approach. A high-level Java profiler with low-overhead can enable even more sophisticated optimization approaches (i.e., thread-grouping based on data dependencies and dynamic NUMA-aware scheduling). Therefore, novel techniques are a necessity in order to improve the efficiency of the new and the already existing Java profilers and prototype novel optimization techniques. For example, sampling techniques can be utilized in order to effectively reduce the profiling overhead. A low-overhead profiler can then adopt sophisticated approaches for accurate online analysis of memory access patterns and locality like Stat-cache [BH05]. In addition, a profiling tool such as PerfUtil, can be improved by enabling the monitoring of additional hardware events. The plethora of hardware events that is provided by

Hardware Performance Counters can enable plenty and even more sophisticated profiling opportunities.

- **Increase Inteligence of the optimization mechanism:** The dynamic optimization mechanism that was presented in Chapter 7 has provided a solid foundation for online dynamic NUMA optimizations in the VM. However, the drawbacks of the current state of the mechanism point towards numerous research opportunities. For example, the intelligence of that mechanism can be improved in order to dynamically adapt in accordance with the behavior of the application in a more effective way. The performance evaluation has shown that some applications (i.e., `xalan`) are not benefited even by mitigating the number of ineffective migrations probably due to different phases that their algorithm comprise. Consequently, the ability of the mechanism to precisely detect the application phases (if any) and dynamically parametrize itself (i.e., the optimization interval) can potentially improve overall performance.

Bibliography

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

[AMD19] AMD. Hypertransport consortium. https://www.hypertransport.org, 2019.

[AMD20] AMD. AMD64 Architecture Programmer's Manual Volume 2 : System Programming, 2020.

[APB+20] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 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*, ASPLOS '20, page 283–300, New York, NY, USA, 2020. Association for Computing Machinery.

[AS15] Khaled Alnowaiser and Jeremy Singer. 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*, LCPC 2015, page 191–205, Berlin, Heidelberg, 2015. Springer-Verlag.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on*

Software Engineering, ICSE '04, page 137–146, USA, 2004. IEEE Computer Society.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, aug 2000.

[BGH+06] 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. 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*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[BH05] Erik Berg and Erik Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 169–180, New York, NY, USA, 2005. Association for Computing Machinery.

[CCL05] Peter Y.K. Cheung, George A. Constantinides, and Wayne Luk. Multiprocessors. In WAI-KAI CHEN, editor, *The Electrical Engineering Handbook*, pages 335–342. Academic Press, Burlington, 2005.

[Che18] Rui Chen. Dacapo 9.12 MR1 Release Notes. https://github.com/dacapobench/dacapobench/blob/468b86874a2f62c66d111fc871674f935619ca0b/benchmarks/RELEASE_NOTES.txt, January 2018.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[Cora] Oracle Corporation. About the Oracle JRockit JDK. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/aboutjrockit.html.

[Corb] Oracle Corporation. Oracle Solaris Studio Software. https://docs.oracle.com/cd/E37069_01/html/E37073/gkmgs.html.

[Cor06] Oracle Corporation. Java Language and Virtual Machine Specifications. https://docs.oracle.com/javase/specs/, 2006.

[Cor08] Intel Corporation. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem) - Whitepaper. https://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf, 2008.

[Cor11a] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual: System Programming Guide, 3B, 2011.

[Cor11b] Oracle Corporation. Java 7 - HotSpot™ Virtual Machine Performance Enhancements. https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html, 2011.

[Cor12a] Jonathan Corbet. Toward better NUMA scheduling. https://lwn.net/Articles/486858/, 2012.

[Cor12b] Intel Corporation. Intel ® Xeon ® Processor E5-2600 Product Family Uncore Performance Monitoring Guide, March 2012.

[Cor13] Jonathan Corbet. NUMA scheduling progress. https://lwn.net/Articles/568870/, 2013.

[Cor14a] Intel Corporation. Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2014.

[Cor14b] Oracle Corporation. Java Fork/Join. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html, March 2014.

[Cor14c] Oracle Corporation. Java Stream. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html, March 2014.

[Cor16] Oracle Corporation. VisualVM - All-in-One Java Troubleshooting Tool. https://visualvm.github.io/, 2016.

[Cor20] Intel Corporation. Intel QuickPath Interconnect: Maximizing multicore processor performance. https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, 2020.

[CSBA17] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 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*, ASPLOS '17, page 207–221, New York, NY, USA, 2017. Association for Computing Machinery.

[CSTL19] Mei-Ling Chiang, Wei-Lun Su, Shu-Wei Tu, and Zhen-Wei Lin. Memory-aware kernel mechanism and policies for improving internode load balancing on NUMA systems. *Software: Practice and Experience*, 49(10):1485–1508, 2019.

[DBSEE13] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 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*, OOPSLA '13, page 355–372, New York, NY, USA, 2013. Association for Computing Machinery.

[DEKB16] Maria Dimakopoulou, Stéphane Eranian, Nectarios Koziris, and Nicholas Bambos. Reliable and Efficient Performance Monitoring in Linux. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16. IEEE Press, 2016.

[DFF+13] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark

Roth. 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*, ASPLOS '13, page 381–394, New York, NY, USA, 2013. Association for Computing Machinery.

[DJL09] Michael Dawson, Graeme Johnson, and Andrew Low. Best practices for using the Java Native Interface. https://developer.ibm.com/articles/j-jni/, July 2009.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, jan 1998.

[Don] Jack Donnell. Java* Performance Profiling using the VTune™ Performance Analyzer. https://www.intel.com/content/dam/develop/external/us/en/documents/219355-vtune-performance-profiling-178112.pdf.

[DS11] John Demme and Simha Sethumadhavan. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. *SIGARCH Comput. Archit. News*, 39(3):353–364, jun 2011.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, page 365–376, New York, NY, USA, 2011. Association for Computing Machinery.

[Ecl] Eclipse. Eclipse OpenJ9. https://www.eclipse.org/openj9/docs/.

[Edg09] Jake Edge. Perfcounters added to the mainline. https://lwn.net/Articles/339361/, jul 2009.

[EGMdB15a] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page, may 2015.

[EGMdB15b] Stephane Eranian, Eric Gouriou, Tipp Moseley, and Willem de Bruijn. Tutorial - Perf Wiki: multiplexing and scaling events. https://perf.wiki.kernel.org/index.php/Tutorial#multiplexing_and_scaling_events, may 2015.

[GH09] James R Goodman and Herbert Hing Jing Hum. MESIF: A two-hop cache coherency protocol for point-to-point interconnects. Technical report, University of Auckland, 2009.

[GK06] Pawel Gepner and Michal Kowalik. Multi-Core Processors: New Way to Achieve High System Performance. In *PARELEC 2006 - Proceedings: International Symposium on Parallel Computing in Electrical Engineering*, pages 9–13, 01 2006.

[GLD+14] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 231–242, USA, 2014. USENIX Association.

[Gro05] HSQLDB Group. MVStore - H2 Database Engine. https://www.h2database.com/html/mvstore.html, December 2005.

[GTS+15] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 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*, ASPLOS '15, page 661–673, New York, NY, USA, 2015. Association for Computing Machinery.

[GTSS13] Lokesh Gidra, Gaël Thomas, Julien Sopena, and Marc Shapiro. 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*, ASPLOS '13, page 229–240, New York, NY, USA, 2013. Association for Computing Machinery.

[Inc09] Lightbend2 Inc. Akka: Build Concurrent, Distributed, and Resilient

Message-Driven Applications for Java and Scala. https://akka.io/, July 2009.

[IO13] Reactors IO. Reactors.IO. http://reactors.io/, 2013.

[IS14] Shams M. Imam and Vivek Sarkar. 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*, AGERE! '14, page 67–80, New York, NY, USA, 2014. Association for Computing Machinery.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[Kam11] Ali Kamali. Sharing Aware Scheduling on Multicore Systems. Master's thesis, Simon Fraser University, Burnaby, Canada, 2011.

[KCR+17] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 74–82, New York, NY, USA, 2017. Association for Computing Machinery.

[Ker10] Michael Kerrisk. *The Linux Programming Interface*. No Starch Press, San Francisco, CA, USA, 2010.

[Kim20] Sangheon Kim. JEP 345: NUMA-Aware Memory Allocation for G1. https://openjdk.org/jeps/345, 2020.

[KMJV12] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A Black-Box Approach to Understanding Concurrency in DaCapo. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 335–354, New York, NY, USA, 2012. Association for Computing Machinery.

[Kot] Jan Kotek. MapDB: database engine. https://github.com/jankotek/mapdb/.

[Lan02] Nick Langley. Write once, run anywhere? https://www.computerweekly.com/feature/Write-once-run-anywhere, May 2002.

[LBM15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 51–62, New York, NY, USA, 2015. Association for Computing Machinery.

[LBMW17] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 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*, ICPE '17, page 3–14, New York, NY, USA, 2017. Association for Computing Machinery.

[LCCAS14] Robert V Lim, David Carrillo-Cisneros, Wail Y Alkowaileet, and Isaac D Scherson. Computationally Efficient Multiplexing of Events on Hardware Counters. *Linux Symposium*, pages 101–110, 2014.

[LQF15] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 277–289, USA, 2015. USENIX Association.

[MG11] Zoltan Majo and Thomas R. Gross. Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, page 11–20, New York, NY, USA, 2011. Association for Computing Machinery.

[MG12] Zoltan Majo and Thomas R. Gross. Matching Memory Access Patterns and Data Placement for NUMA Systems. In *Proceedings*

of the Tenth International Symposium on Code Generation and Optimization, CGO '12, page 230–241, New York, NY, USA, 2012. Association for Computing Machinery.

[MTL21] Ruairidh MacGregor, Phil Trinder, and Hans-Wolfgang Loidl. Improving GHC Haskell NUMA Profiling. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing*, FHPNC 2021, page 1–12, New York, NY, USA, 2021. Association for Computing Machinery.

[NDP17] Richard Neill, Andi Drebes, and Antoniu Pop. Fuse: Accurate Multiplexing of Hardware Performance Counters Across Executions. *ACM Trans. Archit. Code Optim.*, 14(4), dec 2017.

[Neo10] Neo4j. Graph Data Platform — Graph Database Management System — Neo4j. https://neo4j.com/, February 2010.

[OG03] Doug O'flaherty and Michael Goddard. AMD Opteron Processor Benchmarking for Clustered Systems - Whitepaper. https://manualzz.com/doc/29678882/amd-whitepaper--amd-opteron%E2%84%A2-processor-benchmarking-for, 2003.

[Ope] OpenHFT. Chronicle Map. https://github.com/OpenHFT/Chronicle-Map.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[PKD+18] Maria Patrou, Kenneth B. Kent, Gerhard W. Dueck, Charlie Gracie, and Aleksandar Micic. NUMA Awareness: Improving Thread and Memory Management. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 119–123, New York, NY, USA, 2018. IEEE.

[PP84] Mark S. Papamarcos and Janak H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, page 348–354, New York, NY, USA, 1984. Association for Computing Machinery.

[PRL+19] 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. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[PZFK20] Orion Papadakis, Foivos S. Zakkak, Nikos Foutris, and Christos Kotselidis. 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 2020, page 80–88, New York, NY, USA, 2020. Association for Computing Machinery.

[QYMN09] Yuan Qingbo, Bao Yungang, Chen Mingyu, and Sun Ninghui. A Scalability Analysis of the Symmetric Multiprocessing Architecture in Multi-Core System. In *2009 IEEE International Conference on Networking, Architecture, and Storage*, pages 231–234, New York, NY, USA, 2009. IEEE.

[Ram06] R M Ramanathan. Intel Multi-Core Processors: Leading the Next Digital Revolution - Whitepaper. https://courses.e-ce.uth.gr/CE432/voh0hmata/IA32/multi-core-revolution.pdf, 2006.

[RCB16] Andrea Rosà, Lydia Y. Chen, and Walter Binder. AkkaProf: A Profiler for Akka Actors in Parallel and Distributed Applications. In Atsushi Igarashi, editor, *Programming Languages and Systems*, pages 139–147, Cham, 2016. Springer International Publishing.

[RG09] Ian Rogers and Dave Grove. The Strength of Metacircular Virtual Machines: Jikes RVM. In Diomidis Spinelis and Georgios Gousios, editors, *Beautiful Architecture*, chapter 10, pages 235–260. O'Reilly Media, Inc, Sebastopol, CA, USA, 2009.

[RKN+17] Andrey Rodchenko, Christos Kotselidis, Andy Nisbet, Antoniu Pop, and Mikel Lujan. MaxSim: A simulation platform for managed

applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 141–152, New York, NY, USA, 2017. IEEE.

[RRB20] Eduardo Rosales, Andrea Rosà, and Walter Binder. FJProf: Profiling Fork/Join Applications on the Java Virtual Machine. In *Proceedings of the 13th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '20, page 128–135, New York, NY, USA, 2020. Association for Computing Machinery.

[Rus] Einar Rustad. NUMAConnect Whitepaper.

[Shi] Aleksey Shipilev. Java Benchmarking: as easy as two timestamps. https://shipilev.net/talks/jvmls-July2014-benchmarking.pdf.

[Tea10] Scala STM Team. ScalaSTM - Library-Based Software Transactional Memory for Scala. https://nbronson.github.io/scala-stm/intro.html, December 2010.

[Tec] EJ Technologies. JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html.

[TW06] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems - Design and Implementation - Third Edition*. Pearson Education Inc, Upper Saddle River, NJ 07458, 2006.

[Ven96] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., USA, 1996.

[Wea13] Vincent M Weaver. Linux perf_event Features and Overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, number April in FastPath '13, page 80, 2013.

[WHVDV+13] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[YAR09] Rui Yang, Joseph Antony, and Alistair Rendell. Effective Use of Dynamic Page Migration on NUMA Platforms: The Gaussian Chemistry Code on the SunFire X4600M2 System. In *Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, ISPAN '09, page 63–68, USA, 2009. IEEE Computer Society.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 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*, ASPLOS XV, page 129–142, New York, NY, USA, 2010. Association for Computing Machinery.

[ZJ13] Xinghui Zhao and Nadeem Jamali. Load Balancing Non-Uniform Parallel Computations. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, page 97–108, New York, NY, USA, 2013. Association for Computing Machinery.

[ZLR16] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So Many Performance Events, so Little Time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[ZNM+18] Foivos S. Zakkak, Andy Nisbet, John Mawer, Tim Hartley, Nikos Foutris, Orion Papadakis, Andreas Andronikakis, Iain Apreotesei, and Christos Kotselidis. On the Future of Research VMs: A Hardware/Software Perspective. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of*

Programming, Programming'18 Companion, page 51–53, New York, NY, USA, 2018. Association for Computing Machinery.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65, oct 2016.

[ZZG+21] Xin Zhao, Jin Zhou, Hui Guan, Wei Wang, Xu Liu, and Tongping Liu. NumaPerf: Predictive NUMA Profiling. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 52–62, New York, NY, USA, 2021. Association for Computing Machinery.