

Master's Thesis

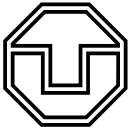
**Integration of Scale-Up Databases into
HARP's Resource Management
Framework**

Leonie Seelisch

16th September 2025

Dresden University of Technology
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Supervising Professor:	Prof. Dr.-Ing. Horst Schirmeier
Referee:	Dr.-Ing. Michael Roitzsch
Supervisor:	Till Smejkal



Task for the preparation of a Master Thesis

Name: Leonie Seelisch
Matriculation number: 4935175
Matriculation year: 2023
Title: Integration of Scale-Up Databases into HARP's Resource Management Framework

Objectives of work

HARP is an existing energy-aware resource management framework designed for heterogeneous multi-core processors. It distinguishes itself from state-of-the-art solutions by enabling fine-grained, application-integrated resource management. This is achieved through two key mechanisms: (1) A generic interface that facilitates bidirectional communication between applications and the resource manager. (2) A structured representation of application configurations, annotated with non-functional characteristics, to guide intelligent resource allocation. HARP can both interpret application-level performance metrics and dynamically adapt runtime parallelism to match available hardware resources.

The goal of this thesis is to integrate a scale-up database system into HARP's management approach. Scale-up databases are characterized by their ability to utilize multiple cores and system resources efficiently within a single machine. To integrate such a database into HARP, the following tasks must be addressed: (1) *Analysis of a Target Database*: Identify and analyze a suitable scale-up database system. The goal is to determine its internal scaling mechanisms (e.g., parallel query execution knobs, thread pool sizing) and locate runtime-accessible performance metrics. (2) *Runtime Adaptation & Monitoring*: Implement mechanisms to extract relevant performance data during execution and feed this information into HARP. In parallel, extend HARP's control to adjust the database's internal parallelism based on the available system resources. (3) *Integration with HARP*: Integrate the adaptation and monitoring mechanisms of the database into HARP's existing resource allocation logic to ensure smooth and effective management. (4) *Evaluation*: Assess the impact of the integration through a series of benchmarks and realistic workload scenarios. Metrics should include performance and resource efficiency of the database following HARP's management decisions.

Optional Extension (Time Permitting): Explore NUMA-awareness within the target database and analyze its implications for HARP's resource allocation strategy. Propose and implement interface extensions that allow HARP to respect NUMA-related constraints, enhancing performance and locality-aware scheduling.

Referee: Dr. Michael Roitzsch
Supervisor: Till Smejkal
Issued on: 16th May 2025
Due date for submission: 17th October 2025

Prof. Dr. Horst Schirmeier
Supervising professor

Statement of authorship

I hereby certify that I have completed this work on my own and without the help of anything other than the indicated sources and resources.

Dresden, September 16, 2025

Leonie Seelisch

Abstract

Heterogeneous systems are designed to enhance performance-per-watt ratios by combining high-performance cores with energy-efficient cores. However, this diversity in core capabilities increases the complexity of scheduling decisions. The HARP framework is a resource management system designed to optimize both performance and energy consumption on heterogeneous multicore processors. HARP manages applications by dynamically allocating resources based on observed workload characteristics and system conditions. Until now, the set of applications supported by HARP include but are not limited to OpenMP and Intel TBB applications.

In this thesis, I expand the set of managed applications by integrating the database management system DuckDB into HARP. This integration involves adapting DuckDB’s internal parallelism in response to HARP’s resource allocations and providing database-specific performance metrics to guide the framework’s scheduling decisions.

Furthermore, I extend HARP with NUMA awareness to enable topology-aware scheduling and data placement by adapting the resource allocation algorithm to consider NUMA topology and introducing a data migration feature to maintain memory locality.

The evaluation of the integration of DuckDB into HARP indicates that, when HARP is provided with pre-generated performance and energy data, single DuckDB programs experience only a minor geometric mean performance degradation of 1.7 % and a slight increase in energy consumption of 0.4 %. In multi-application scenarios, HARP demonstrates significant benefits, with mean improvements of 21 % in performance and 28 % in energy efficiency, emphasizing the effectiveness of the integration under concurrent workloads. When relying solely on online measurements, single-program execution incurs a substantial mean performance drop of 61 % and an energy increase of 10 %, while multi-application execution still yields energy savings of 21 % despite a 19 % performance decrease. Independently of the DuckDB integration, the NUMA-aware extension of HARP achieves a mean performance gain of 15.5 % over baseline scheduling, underscoring the benefits of topology-aware resource management in heterogeneous systems.

Contents

List of Figures	XI
List of Tables	XIII
Listings	XV
1 Introduction	1
2 Technical Background	3
2.1 HARP	3
2.1.1 The challenge of heterogeneity	3
2.1.2 General structure and communication	4
2.1.3 Resource Allocation Algorithm	6
2.1.4 Application adaptivity	7
2.1.5 MappingFeature class	8
2.2 Scale-Up Databases	8
2.2.1 Scale-up vs. scale-out	8
2.2.2 Choosing a suitable DBMS	9
2.2.3 Properties of DuckDB	9
2.3 NUMA	10
2.3.1 Motivation for NUMA architectures	10
2.3.2 Further subdivisions	12
2.3.3 Extending HARP	13
2.4 Related Work	14
2.4.1 Database scaling	14
2.4.2 NUMA awareness	15
3 DuckDB Scaling	19
3.1 Design	19
3.1.1 Integration with HARP	19
3.1.2 Identifying integration points	21
3.2 Implementation	22
3.2.1 Scaling with DuckDB	23
3.2.2 Providing feedback to the HARP RM	25
4 NUMA Awareness	29
4.1 Design	29
4.1.1 NUMA topology data	29
4.1.2 Integrating NUMA information into HARP’s scheduling decisions	31

4.1.3	Approximating performance loss	31
4.1.4	Thread migration across NUMA nodes	32
4.2	Implementation	33
4.2.1	Resource allocation adaptations	34
4.2.2	Approximating performance loss	38
4.2.3	MigratePages feature	42
5	Evaluation	43
5.1	Integration of DuckDB into HARP	43
5.1.1	Measurement setup	43
5.1.2	Benchmarks	43
5.1.3	Experimental results	44
5.2	NUMA Awareness Extension	51
5.2.1	Measurement setup	51
5.2.2	Benchmarks	51
5.2.3	Experimental results	51
6	Conclusion and Future Work	63
6.1	Conclusion	63
6.2	Future Work	64
	Bibliography	65
A	Additional Results	69
A.1	Full Table of Performance Loss Measurements	70
A.2	Complete Measurements of the Evaluation of Scaling DuckDB	71

List of Figures

2.1	Overview of the HARP framework, focussing on general structure and communication. Taken from the Dissertation of Till Smejkal [44] and adapted for this thesis.	4
2.2	Conceptual representation of a NUMA system with two sockets, each containing 12 cores organized into a NUMA node. A core is denoted as c_i , where i is a unique identifier. Each socket has its own memory controller (MC) that provides access to locally attached memory with a NUMA distance of 10. Memory accesses across sockets traverse the inter-socket interconnect and have a NUMA distance of 20.	11
2.3	Conceptual representation of a NUMA system with two sockets, each subdivided into two SNCs, each containing six cores. A core is denoted as c_i , where i is a unique identifier. Each SNC has its own memory controller (MC) that provides access to locally attached memory with a NUMA distance of 10. Memory accesses across SNCs within the same socket traverse the intra-socket interconnect and have a NUMA distance of 12. Memory accesses across sockets traverse the inter-socket interconnect and have a NUMA distance of 20.	13
4.1	Conceptual representation of a NUMA system with two sockets, each subdivided into two SNCs, each containing six cores. The figure shows three applications A , B , and C , each allocated eight cores.	37
4.2	Performance loss (%) of the RA program depending on its memory heaviness, measured in LLC misses per 10^3 instructions	40
5.1	Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and rows per second as performance metric.	45
5.2	Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and IPS as performance metric.	48
5.3	Relative improvement in % of HARP over CFS. HARP runs with online-generated operating points and rows per second as performance metric.	49
5.4	Thread-to-core assignments for the 2-Apps scenario managed by EEVDF	53
5.5	Thread-to-core assignments for the 2-Apps scenario managed by HARP without NUMA awareness	53
5.6	Thread-to-core assignments for the 2-Apps scenario managed by HARP with NUMA awareness	54
5.7	Thread-to-core assignments for the 8-Apps scenario managed by EEVDF	56
5.8	Thread-to-core assignments for the 8-Apps scenario managed by HARP without NUMA awareness	56

5.9	Thread-to-core assignments for the 8-Apps scenario managed by HARP with NUMA awareness	57
5.10	Thread-to-core assignments for the 3-Apps scenario managed by EEVDF	58
5.11	Thread-to-core assignments for the 3-Apps scenario managed by HARP without NUMA awareness	58
5.12	Thread-to-core assignments for the 3-Apps scenario managed by HARP with NUMA awareness	59
5.13	Thread-to-core assignments for the 12-Apps scenario managed by EEVDF	60
5.14	Thread-to-core assignments for the 12-Apps scenario managed by HARP without NUMA awareness	60
5.15	Thread-to-core assignments for the 12-Apps scenario managed by HARP with NUMA awareness	61
5.16	Absolute execution time for each scenario scheduled with EEVDF, NUMA-unaware HARP, and NUMA-aware HARP	62
A.1	Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and rows per second as performance metric.	71
A.2	Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and instructions per second as performance metric.	72
A.3	Relative improvement in % of HARP over CFS. HARP runs with online-generated operating points and rows per second as performance metric.	73

List of Tables

4.1	Performance loss (%) of different programs depending on their memory heaviness, measured in LLC misses per 10^3 instructions	40
A.1	Performance loss (%) of different programs depending on their memory heaviness, measured in LLC misses per 10^3 instructions	70

Listings

2.1	Simplified hardware description file for the Intel Raptor Lake Core i9-13900K	5
2.2	Structure of an Operating Point	5
3.1	Example for a simple DuckDB program	21
3.2	Pseudo code for adapting DuckDB's internal parallelism through the <code>ScalableApplication</code> feature	24
4.1	Example hardware description file extension for a system with two Intel Xeon Max 9468 CPUs	30
4.2	Adapting Operating Point measurements	35
4.3	Pseudo code for adapting the LR-based approximation algorithm	36
4.4	Pseudo code for adapting the LR-based approximation algorithm	37

1 Introduction

In recent years, computer architectures have undergone a profound shift. The steady increase in clock frequencies that once fueled performance gains has reached physical and energy-related limits. As a result, the computing industry has turned to multicore processors as the primary means of improving throughput. Initially, these processors were homogeneous, with identical cores distributed across the chip. However, to optimize the amount of computational work achievable for a given amount of consumed energy, heterogeneous CPUs have emerged, combining cores of different types and capabilities within the same processor package. Modern architectures such as ARM’s big.LITTLE [20], Intel’s hybrid x86 [24] and Apple M-series [2] designs exemplify this trend, featuring high-performance cores alongside energy-efficient cores.

While heterogeneous CPUs promise substantial efficiency improvements, they also complicate resource management. Assigning threads to cores of different types requires careful consideration. Allocating latency-sensitive tasks to energy-efficient cores may degrade performance, while indiscriminate use of high-performance cores can waste energy. The complexity is further amplified in Non-Uniform Memory Access (NUMA) systems, where memory access latency depends on the topology of cores and memory controllers. Poor placement of threads and data can result in costly remote memory accesses, undermining both performance and efficiency. Consequently, effective scheduling strategies are critical to harnessing the full potential of heterogeneous and NUMA-based architectures.

Heterogeneity-aware Adaptive Resource Partitioning (HARP) [45] has been developed as a framework to address the challenges posed by heterogeneous systems. HARP is characterized by its advanced resource allocation algorithm, which continuously monitors both performance and energy consumption to dynamically adapt to application behavior and varying system conditions. This algorithm operates in combination with a bidirectional communication interface between the framework and its managed applications, enabling HARP to coordinate resource assignments while also receiving performance feedback to guide its scheduling decisions.

Within this thesis, I examine the integration of a database management system (DBMS) into HARP as one interesting direction. Databases represent a particularly relevant class of workloads for scale-up systems, as they are both resource-intensive and highly sensitive to scheduling and parallelization strategies. For this purpose, I select the analytical in-memory DBMS DuckDB [16] as a representative system to explore its integration into HARP.

Another key aspect I explore in this thesis is the incorporation of NUMA awareness into HARP’s scheduling algorithm. As previous research has shown, NUMA effects can substantially impact application performance due to differences in local and remote memory access latencies. Most NUMA-focused projects, however, concentrate solely

on NUMA placement and are implemented as standalone systems that aim primarily at performance maximization [19, 21, 27, 37, 53]. In contrast, HARP already provides a comprehensive framework that balances performance and energy efficiency across heterogeneous hardware. Integrating NUMA awareness into HARP therefore offers the opportunity to combine topology-sensitive placement with energy-aware scheduling in a unified system.

The contributions of this thesis can be divided into two main directions. The first concerns the integration of DuckDB into HARP. This part of the work identifies suitable adaptation points within DuckDB’s internal threading mechanisms and demonstrates how HARP can adjust database parallelism. In addition, I develop mechanisms to extract application-specific performance metrics from DuckDB and make them available to HARP in order to guide its scheduling decisions.

The second contribution extends HARP with NUMA awareness. To reach this, I adapt HARP’s resource allocation algorithm to take NUMA topology into account, enabling placement strategies that distinguish between different topology levels. To further improve memory locality, I introduce a data migration feature that allows applications to move allocated pages when threads are reassigned across NUMA nodes. In this way, the extension combines NUMA-aware placement with HARP’s original goal of optimizing both performance and energy efficiency.

The remainder of this thesis is structured as follows. Chapter 2 provides the fundamental background necessary for this work, presenting the HARP framework in detail, outlining its architecture, communication model, and resource allocation mechanisms. In addition, it introduces core concepts related to scale-up DBMSs and NUMA architectures, and it examines relevant related work in these areas. Chapter 3 discusses the integration of DuckDB into HARP, beginning with an analysis of the database’s internal structure, continuing with the identification of adaptation points, and concluding with the development of mechanisms for reporting application-specific performance metrics. Chapter 4 presents the NUMA-aware extension of HARP, detailing the mechanisms for exposing system topology information to HARP, the adaptations of the allocation algorithm to leverage this information, and the support for data migration. Chapter 5 provides an experimental evaluation of both the integration of DuckDB and the NUMA-aware extension, with a focus on performance and energy efficiency across a variety of workloads. Finally, Chapter 6 concludes the thesis with a summary of the main findings and an outlook on directions for future research.

2 Technical Background

This chapter provides a fundamental understanding of the key concepts and technologies that form the basis of the research in this thesis. To establish a coherent context for the proposed contributions, I introduce three core areas. First, I outline HARP [45] — an energy-aware resource management system for heterogeneous processors, which provides the foundational framework for this research. Second, I present scale-up *Database Management Systems* (DBMSs) as a new use case for integration into HARP. Third, I introduce NUMA awareness, as I extend HARP to support NUMA-aware resource management as part of this work. Finally, I provide an overview of related work to situate this research within the broader context of existing literature.

2.1 HARP

HARP (Heterogeneity-aware Addaptive Resource Partitioning) is a resource management framework integrated into the Linux operating system and designed to improve the execution of applications on modern heterogeneous computing systems. It optimizes performance and energy efficiency by dynamically adapting resource allocations based on workload characteristics and hardware architecture. The following section discusses the core mechanisms and design principles underlying HARP.

2.1.1 The challenge of heterogeneity

Heterogeneous processors integrate different types of processing cores within a single system, each optimized for different performance and energy characteristics. This architectural design enables flexible and energy-efficient execution of diverse workloads by assigning tasks to the most suitable core types. A prominent example is ARM’s big.LITTLE architecture, which combines powerful *big* cores with power-saving *LITTLE* cores in mobile and embedded systems [20]. Similarly, Intel’s hybrid architecture, as introduced in the Alder Lake and Raptor Lake processor families, adopts a comparable strategy, combining high-performance cores (*P-cores*) with energy-efficient cores (*E-cores*) [24].

However, not all applications benefit equally from this heterogeneous core structure. The actual runtime behavior of an application plays a crucial role in determining whether it can effectively utilize the higher performance of P-cores or whether it is constrained by other factors such as memory bandwidth, I/O latency, or limited parallelism. For example, compute-bound workloads with high instruction-level parallelism may saturate P-cores effectively, while I/O-bound or memory-constrained applications may derive little to no benefit from faster cores and would be better suited to run on E-cores to save energy without sacrificing performance.

Despite this complexity, cost-based thread allocation strategies, which dominate state-of-the-art schedulers, typically rely on static heuristics or simplistic cost models. On Linux, depending on the kernel version, the default scheduler employs either the *Completely Fair Scheduler* (CFS) [11] or the *Earliest Eligible Virtual Deadline First* (EEVDF) [12] algorithm, both of which exemplify this reliance on simplified allocation mechanisms. These approaches often fail to account for dynamic application behavior and the architectural asymmetry of modern heterogeneous processors. As a result, they do not fully exploit the performance and energy-saving potential offered by these systems, leading to inefficient core utilization and suboptimal system performance.

To address these limitations, HARP provides an advanced framework for resource management on heterogeneous processors. It implements a sophisticated resource allocation algorithm that dynamically adapts to application behavior and system conditions. Moreover, it provides a communication interface linking applications with the HARP framework. This allows HARP to notify applications of resource allocation decisions while enabling applications to convey their current resource needs and performance characteristics back to the framework. Together, these mechanisms enable HARP to tailor resource assignments to the performance and energy characteristics of both the workload and the underlying hardware.

2.1.2 General structure and communication

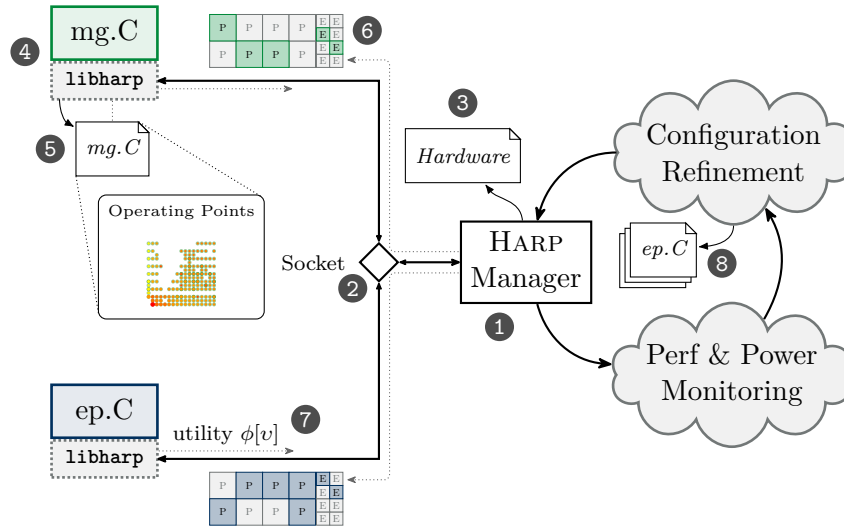


Figure 2.1: Overview of the HARP framework, focussing on general structure and communication. Taken from the Dissertation of Till Smejkal [44] and adapted for this thesis.

The general structure of the HARP framework is illustrated in Figure 2.1. The central component of HARP is the *Resource Manager* (RM) (1), which is implemented as a single instance responsible for monitoring and managing all applications, also referred

to as *clients*, running under its control. Clients connect to the RM via a socket-based interface (②) to enable resource management.

Figure 2.1 illustrates `ep.C` and `mg.C`, two *OpenMP*-based [6] example applications from the *NAS Parallel Benchmarks* (NPB) [3], which are managed by HARP. To facilitate the communication and coordination between the RM and individual applications, HARP provides an auxiliary library called `libharp`. For every application, a separate `libharp` instance is used to act as an intermediary between the RM and the application (④). When the RM makes a resource allocation decision — that is, when it determines which application threads should be scheduled on which processor cores (⑥) — it communicates this decision via `libharp`. The library then enforces this new allocation for the corresponding application.

```
core_types:
  - {type: "P-core", threads: 2, power_coefficient: 8}
  - {type: "E-core", threads: 1, power_coefficient: 3}

equivalence_scheme: "core-type"
energy_measure_method: "perf"

cores:
  - type: "P-core"
    threads:
      - {name: "P0_0", affinity: 0}
      - {name: "P0_1", affinity: 1}
  - type: "E-core"
    threads:
      - {name: "E00", affinity: 16}
```

Listing 2.1: Simplified hardware description file for the Intel Raptor Lake Core i9-13900K

```
struct OperatingPoint {
    struct Configuration {
        std::string name;                // Unique name
        CPUThreadSet threads;           // Active CPU threads
        std::map<std::string, int> core_counts; // Core counts by type
    };

    struct Metrics {
        double utility;                // Performance (IPS)
        double power;                  // Power consumption
    };
}
```

Listing 2.2: Structure of an Operating Point

To make informed scheduling and allocation decisions, the RM relies on two essential sources of information: a *hardware description file* (③) and application-specific *Operating Points* (OPs) (⑤). The hardware description file contains detailed information about the underlying system architecture, specifying e.g. available cores as well as their respective

classifications into P-cores and E-cores. An example of such a file is shown in Listing 2.1. An OP, on the other hand, represents a specific configuration together with its performance and energy consumption. As illustrated in Listing 2.2, the configuration defines how many threads are used and how many cores of each type are allocated, while the associated metrics provide insight into the application’s performance and energy consumption at that specific configuration.

Such OPs can be provided in two ways. First, OPs can be pre-generated and supplied to the RM during application registration through a so-called *mapping file*, as exemplified by the `mg.C` application in Figure 2.1 (5). Alternatively, OPs can be generated dynamically at runtime, as shown for `ep.C`. In this case, HARP executes the application under different configurations and continuously monitors performance and energy measurements (8). For each configuration, the observed metrics are stored in a corresponding OP, allowing the RM to capture the application’s performance and energy behavior under varying resource allocations.

By default, HARP measures *Instructions Per Second* (IPS) as a generic performance metric to estimate how much useful work an application performs under a given configuration. However, HARP also supports the integration of application-specific utility metrics (7), which can be provided by the application itself via `libharp`. Chapter 3 provides a detailed discussion illustrating possible examples of such custom utility metrics.

By combining information of the underlying hardware with application-specific OPs, the RM can evaluate trade-offs between computational efficiency and energy usage to guide resource allocation decisions with the aim to maximize overall system performance while minimizing total energy consumption.

2.1.3 Resource Allocation Algorithm

The primary goal of HARP is to improve overall system performance and energy efficiency. To achieve this, for the current set of managed applications, it must determine resource allocations that provide together an optimal trade-off between performance and energy consumption. This task is handled by the *Resource Allocation Algorithm* (RAA), which is the core component responsible for computing suitable allocations.

The RAA can be divided into two main parts. The first part is solving an optimization problem, which involves finding a resource allocation for each application such that the combination of all allocations maximizes overall system performance while minimizing total energy consumption, constrained by the available hardware resources. The second part is determining concrete thread-to-core assignments based on the selected resource allocations.

To solve the optimization problem, the RAA employs an approximation algorithm based on *Lagrangian Relaxation* (LR) [18]. This approximation algorithm receives the set of clients to consider and, for each client, determines the set of OPs that are relevant for the optimization. For each client individually, the RAA selects the OP that achieves the best trade-off between performance and energy consumption. Once an optimal OP has been chosen for every client, the algorithm evaluates the total resource requirements, i.e., the number of cores of each type needed across all applications.

If the total resource demand exceeds the available hardware, the algorithm is rerun with adjustments that bias the OP selection process toward configurations consuming fewer of the over-demanded resources. For example, if too many applications require P-cores, penalties are applied to these cores, and the algorithm iteratively increases these penalties until a feasible solution is found. If the LR algorithm fails to produce a feasible solution within a predefined number of iterations, HARP temporarily allows multiple threads to share cores.

Once a feasible resource allocation is determined, each client has an associated OP specifying how many cores of each type should be assigned. Using this information, together with details about the types of available cores, HARP performs thread-to-core assignments, allocating each client's threads to free cores of the required type.

2.1.4 Application adaptivity

Applications managed by HARP can exhibit varying degrees of runtime adaptivity, which are classified into three categories: *static*, *scalable*, and *custom* applications. Each category corresponds to a different integration strategy within the HARP framework, particularly through its interaction layer `libharp`.

Static applications do not implement any internal mechanisms for runtime adaptation. For these applications, HARP enforces resource allocations by intercepting thread-related POSIX functions such as `pthread_create()` and `pthread_setaffinity_np()` [4]. Thread affinities are dynamically controlled using the `sched_setaffinity()` system call, allowing the RM to constrain threads to specific processor cores based on the current resource assignment.

Scalable applications are programs in which the degree of parallelization can be adjusted dynamically at runtime. While libraries such as OpenMP provide mechanisms for data-level parallelism, the number of threads is typically determined at program startup and remains fixed throughout execution. Such a static configuration may become inefficient if the available computational resources are reallocated during runtime, which may happen under HARP's dynamic resource management.

If the RM provides more cores than the application has configured threads for, the program cannot make use of the additional cores, resulting in underutilization of resources. Conversely, if fewer cores are allocated than the number of active threads, multiple threads must share individual cores. In such cases, the operating system scheduler must time-multiplex these threads, which leads to increased context-switching overhead and potential performance degradation [22].

To address this issue, `libharp` extends applications linked against supported parallelism libraries by adapting their parallelization degree at runtime. It achieves this by intercepting internal library functions that control the number of active threads and adjusting their configuration according to the resources currently assigned by the RM. In the case of OpenMP, `libharp` hooks into the `GOMP_parallel()` function and modifies its `num_threads` parameter at runtime to align with the updated resource allocation.

Custom applications provide greater flexibility for implementing application-specific adaptations, rather than focusing primarily on adjusting the degree of parallelization. Such adaptations may, for example, involve dynamically selecting algorithms based on the characteristics of the currently assigned cores. If an application provides multiple algorithms to accomplish the same task, it can select the one that aligns most effectively with the characteristics of the allocated cores.

Another potential adaptation involves migrating allocated data when an application is reassigned to different cores, in order to maintain data locality and reduce performance degradation. This aspect will be examined in more detail in Chapter 4.

2.1.5 MappingFeature class

To enable applications to implement custom adaptations for tailored optimizations, HARP provides a **MappingFeature** interface. This interface exposes two key functions that allow applications to influence resource allocation and execution behavior.

The first function, **extend_mapping()**, is called when an application has previously generated OPs and allows the application to modify or extend these OPs before they are submitted to the HARP RM, enabling custom optimization strategies to be incorporated directly into the scheduling process.

The second function, **mapping_update()**, is invoked by **libharp** whenever new resource allocations are received from the RM. In this call, **libharp** provides the application with concrete information about the new mapping, including the currently selected OP with details on the number and types of cores assigned. Within the **mapping_update()** function, the application can decide how to respond to the new allocation, for instance by selecting the most suitable algorithm based on the types of cores currently available.

2.2 Scale-Up Databases

The following section presents scale-up DBMSs as a new use case for heterogeneity-aware resource management with HARP. Scale-up DBMSs are designed to efficiently utilize the large memory and multicore capabilities of modern servers, making them suitable candidates for evaluating and extending HARP’s resource management capabilities.

2.2.1 Scale-up vs. scale-out

In parallel and distributed computing, scale-up and scale-out represent two distinct strategies for increasing computational capacity [47]. Scale-up refers to improving performance by utilizing more powerful hardware within a single machine — such as adding CPUs, memory, or specialized accelerators. In contrast, scale-out achieves growth by distributing workloads across multiple interconnected systems, often using commodity hardware.

In the context of DBMSs, scale-out databases are designed for horizontal scaling, often used in cloud or distributed environments where data is partitioned across multiple servers. These systems emphasize fault tolerance and distributed query execution. Scale-up databases, on the other hand, are optimized to exploit the full capabilities of a

single server with many cores, large memory, and often *Non-Uniform Memory Access* (NUMA) topologies [52]. They rely on efficient intra-node parallelism and shared memory architectures.

For HARP, which is specifically designed to manage resources in heterogeneous, multi-core environments, scale-up databases represent an especially relevant use case. Their performance depends critically on effective core assignment, NUMA-aware scheduling, and energy-efficient resource utilization.

2.2.2 Choosing a suitable DBMS

A core requirement of this work is the integration of a DBMS that fits the architectural and operational principles of HARP. Since HARP focuses on resource management of a single machine, the database should be able to exploit multicore scale-up systems, support dynamic adaptation, and allow low-level control and instrumentation.

While a wide variety of scale-up DBMSs exists, many known systems — including *Oracle* [34], *SAP HANA* [41], and *Microsoft SQL Server* [31] — are closed-source. As such, they are difficult to be instrumented or modified at the system level, which rules them out for integration into research-oriented frameworks like HARP in the frame of this thesis. Even some widely used open-source systems pose architectural limitations. For example, *PostgreSQL* [36], though open and extensible, follows a process-based parallelism model. As outlined in Section 2.1.4, HARP relies on intercepting POSIX thread functions to enforce resource allocations. PostgreSQL’s use of processes instead of threads makes this form of fine-grained scheduling infeasible, thereby limiting its applicability.

Given the outlined criteria, *DuckDB* [16] emerged as a suitable choice. DuckDB is an open-source, C++-based analytical DBMS that is well-suited for scale-up architectures. It supports multithreaded execution, allowing HARP’s resource allocation decisions to directly affect the number of threads involved in query execution. Furthermore, since DuckDB and HARP use the same programming language, integrating the communication and management via HARP’s interfaces poses fewer challenges and speeds-up the integration process. Additionally, DuckDB provides built-in profiling functionality for queries and operators, offering readily available performance insights that can be used to supply application-specific feedback to HARP’s RM, as further discussed in Chapter 3.

In summary, DuckDB offers the required openness, adaptability, and technical compatibility for the integration into HARP and serves as a representative scale-up DBMS for exploring resource-aware execution strategies.

2.2.3 Properties of DuckDB

After outlining the motivation for selecting DuckDB as the foundation for this work, I now provide a brief introduction to its key properties and features.

DuckDB is an in-process analytical DBMS designed for *OLAP*-style workloads, i.e., analytical processing scenarios that involve complex, relatively long-running queries operating over substantial portions of the stored dataset [17]. Unlike traditional server-based DBMSs, DuckDB runs entirely within the host process, eliminating client-server

communication overhead and making it suitable both as a standalone analytical engine and as an embedded component within larger systems.

Technically, DuckDB employs a columnar storage format and a vectorized query execution engine that processes data in batches rather than row by row. Moreover, the system supports parallel query execution using a task-based scheduling model that distributes queries into finer-grained tasks executed by multiple worker threads.

DuckDB provides full SQL compliance and supports advanced query optimization techniques. Furthermore, it natively supports modern data formats such as Parquet, CSV, JSON, and Apache Arrow, enabling efficient interoperability with existing data lakes and analytical ecosystems.

In summary, DuckDB combines lightweight deployment, high analytical performance, and seamless integration capabilities, making it a suitable foundation for the integration in HARP.

2.3 NUMA

While HARP is well-suited for managing resources on heterogeneous systems by dynamically adapting thread allocations to core characteristics, it lacks awareness of memory topology and access locality. In particular, it does not consider the effects of *Non-Uniform Memory Access* (NUMA) [42], which can significantly influence application performance on multi-socket or multi-die architectures. To address this limitation, I extended HARP to support NUMA-aware resource management in the context of this work. This section provides the necessary background on NUMA architectures and the motivation for integrating NUMA awareness into HARP.

2.3.1 Motivation for NUMA architectures

Modern server architectures are designed to meet the increasing demands for computational performance and scalability. To achieve this, such systems are typically equipped not only with multiple CPU cores, but also with multiple physical processors [50] mounted to different sockets on a single motherboard. While this parallelism provides significant potential for performance improvements, it also introduces substantial challenges related to memory access and data locality.

2.3.1.1 Memory Wall

One of the fundamental limitations in modern computing systems is the discrepancy between CPU processing speed and main memory latency [52]. CPU cores have become increasingly faster over the years, but memory access times have not kept pace. As a result, CPUs frequently stall while waiting for data to arrive from main memory.

In addition to this inherent latency gap, memory contention arises when multiple cores attempt to access the same memory resources simultaneously. This occurs because the memory controller and DRAM channels can handle only a limited number of concurrent requests. When several cores issue requests at the same time, some requests must wait, further increasing effective memory access latency and causing CPU cores to stall more

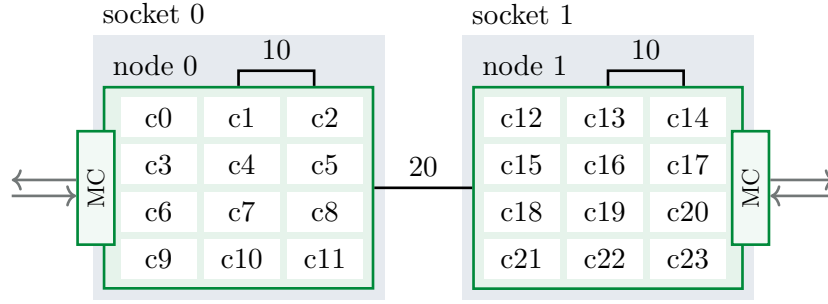


Figure 2.2: Conceptual representation of a NUMA system with two sockets, each containing 12 cores organized into a NUMA node. A core is denoted as c_i , where i is a unique identifier. Each socket has its own memory controller (MC) that provides access to locally attached memory with a NUMA distance of 10. Memory accesses across sockets traverse the inter-socket interconnect and have a NUMA distance of 20.

frequently. Memory contention is already present in single-socket systems, where all cores share the same memory controller and channels, but it becomes even more severe in multi-socket systems, where several processors may attempt to access the same memory resources simultaneously, further increasing contention and memory starvation.

To address the challenges, NUMA architectures were introduced. In NUMA systems, the physical memory is divided into multiple *nodes*, with each node directly attached to a specific socket. Each node comprises a set of CPU cores that share access to a specific region of local memory. This design allows cores to have *local* access to memory within its own NUMA node with low latency, minimizing stalls due to slow memory accesses. Although NUMA systems primarily optimize for local memory access, accessing memory attached to a different socket, referred to as *remote* access, remains possible but typically incurs higher latencies due to inter-processor communication over the interconnect.

In addition to reducing access latencies for local memory, the NUMA design alleviates memory contention by providing each processor with dedicated resources, such as memory controllers and memory channels. This distribution of memory resources reduces competition among processors for the same memory paths, improving overall throughput and scalability in multi-socket systems.

Figure 2.2 shows a conceptual representation of a NUMA system with two sockets, each with 12 cores. Each processor has its own dedicated memory controller, which it uses to access the memory locally attached to it.

The relative proximity between NUMA nodes is often expressed using a NUMA *distance* [42], which quantifies the cost — typically in terms of latency — of accessing memory on a different node. In the example presented in Figure 2.2, a reference value of 10 is used for local memory access, so the distance of 20 to the other node indicates that access to this remote memory is approximately twice as slow. In general, lower distances indicate closer nodes with faster memory access, while higher distances imply more costly remote access. This variation in access latency across nodes is precisely what gives rise to the term *Non-Uniform Memory Access*.

2.3.1.2 ccNUMA

To optimize performance, most modern processors employ multi-level caches, which store frequently used data close to the cores. Consequently, nearly all current systems that feature NUMA architectures implement *cache-coherent* NUMA (ccNUMA) designs, where hardware mechanisms ensure that cached data remains consistent across processors [52]. However, maintaining such cache coherence introduces significant overhead. In ccNUMA systems, inter-processor communication between cache controllers is required to synchronize data whenever multiple caches store the same memory location. While these mechanisms guarantee a consistent memory image, performance can degrade substantially when multiple processors frequently access the same memory regions in rapid succession, as the cost of maintaining coherence grows with contention.

2.3.2 Further subdivisions

In recent years, the computational demands of modern applications have grown significantly. Workloads in areas such as artificial intelligence, hyperscale data centers, and autonomous vehicles increasingly require massive parallelism, high memory bandwidth, and low-latency data access. Traditional monolithic System-on-Chip designs, in which all cores, caches, and interconnects are fabricated on a single large silicon die, are reaching physical and economic limitations [43]. Manufacturing very large monolithic dies reduces yield, increases production cost, and restricts scalability, making it increasingly difficult to satisfy the performance requirements of emerging workloads. To address these challenges, major vendors have adopted different architectural strategies that are outlined in the following sections.

2.3.2.1 Chiplets

Modern processor architectures of AMD are shifting toward multi-die systems, in which multiple smaller silicon dies - also referred to as *chiplets* — are integrated into a single processor. A die represents an individual piece of silicon containing processor cores, caches, and interconnects, and each die operates as a distinct compute unit within the overall processor. By interconnecting multiple dies within the same processor, chiplet-based designs improve manufacturing yields, enable heterogeneous integration of different core types, and allow vendors to scale processor performance more flexibly. Importantly, in such systems, all main memory remains logically visible and accessible from any die within a processor package as well as across processors, but the latency of memory access still depends strongly on the physical data path.

2.3.2.2 Sub-NUMA Clusters

Intel, in contrast, addresses similar challenges within a monolithic die through the concept of Sub-NUMA Clusters (SNCs) [23], which logically subdivide a physical die into smaller clusters, each of which has its own local memory region and memory controller. Similar to the principal of NUMA nodes introduced in Section 2.3.1.1, each SNC comprises a set of cores that share access to a designated region of local memory, which can be

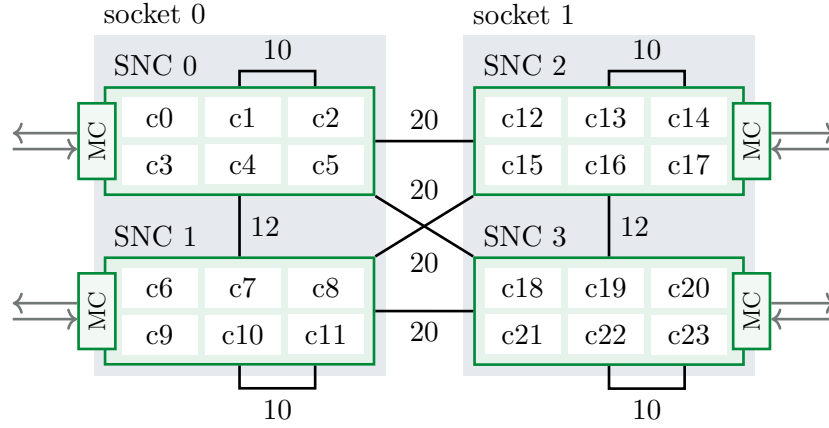


Figure 2.3: Conceptual representation of a NUMA system with two sockets, each subdivided into two SNCs, each containing six cores. A core is denoted as c_i , where i is a unique identifier. Each SNC has its own memory controller (MC) that provides access to locally attached memory with a NUMA distance of 10. Memory accesses across SNCs within the same socket traverse the intra-socket interconnect and have a NUMA distance of 12. Memory accesses across sockets traverse the inter-socket interconnect and have a NUMA distance of 20.

accessed with minimal latency. When a core accesses memory located on a different SNC within the same processor, latency increases slightly because requests must traverse additional on-die interconnects. Accessing memory attached to a different processor still introduces the highest latency, as communication must cross socket boundaries and traverse inter-processor interconnects. Figure 2.3 illustrates a NUMA system consisting of two sockets, where each socket is subdivided into two SNCs, each containing six cores. The varying memory access latencies are represented through NUMA distances. Local memory accesses within the same SNC are assigned a distance of 10, while accesses to a different SNC within the same socket have a slightly higher distance of 12, reflecting an approximate 20% increase in latency compared to local memory access. Cross-socket memory accesses are assigned a distance of 20, representing latency that is approximately twice as high.

In the context of SNCs and chiplets, the traditional notion of a NUMA node — defined as a group of processor cores sharing access to a specific local memory region — shifts to reflect finer-grained divisions. Each SNC or chiplet can be treated as a distinct NUMA node. In the following, references to effects across NUMA nodes correspond to interactions between SNCs or chiplets, while instances involving cross-socket effects are explicitly indicated.

2.3.3 Extending HARP

As discussed in Section 2.3.1.1 and Section 2.3.1.2, NUMA architectures can exhibit degraded performance both when memory accesses involve distant nodes, incurring higher

latency, and when multiple NUMA nodes frequently access the same memory region, due to cache coherence overheads. To mitigate these effects, efficient scheduling and resource allocation strategies need to take the NUMA topology into account. In general, cores and memory should be allocated in a NUMA-friendly manner to minimize cross-NUMA communication and reduce the overhead caused by maintaining cache coherence [52]. Aligning thread placement and memory allocation with the underlying hardware topology enables systems to exploit locality more effectively, which can significantly improve performance and scalability in multi-socket and multi-die environments. For this reason, it is desirable for HARP to incorporate NUMA awareness into its resource allocation and scheduling decisions.

2.4 Related Work

In the following sections, I examine selected studies that address challenges similar to those targeted in this thesis. This includes, on the one hand, strategies for scaling DBMSs to handle increasing workloads efficiently, and on the other hand, approaches for integrating NUMA awareness to optimize thread and data placement.

2.4.1 Database scaling

Scaling database systems to efficiently handle increasing workloads is a well-studied challenge, and a variety of approaches have been proposed to address it. In the course of this work, I examined several representative projects: some target cloud environments [10, 26, 33], one presents a hardware-centric approach to scale databases [5], and another is tailored specifically toward optimizing the execution of small queries [35]. For the scope of this thesis, however, I concentrate on two approaches that best align with the objectives of my work. These two strategies illustrate distinct methodologies for workload management and query scheduling in scale-up databases. I analyze their core mechanisms and performance characteristics, emphasizing how they differ from the integration of DuckDB into HARP in terms of feature utilization, training requirements, and adaptability.

2.4.1.1 Task- and morsel-based scheduler

Wagner et al. [49] introduce a task- and morsel-based scheduler for in-memory analytical databases. Their design dynamically decomposes queries into task sets and morsels at runtime and is implemented within the Umbra DBMS. In this context, a morsel is defined as a discrete batch of tuples within a query pipeline and serves as the smallest unit of work during execution and every task consists of an arbitrary number of morsels. While DuckDB also employs internal task scheduling, it does not decompose queries into such fine-grained morsels, limiting the scheduler’s ability to adapt to varying task sizes.

Since the introduction of morsels causes high variance in task granularity, Wagner et al. employ adaptive morsel execution, dynamically adjusting morsel sizes to balance responsiveness and scheduling overhead. Additionally, adaptive query priorities transparently favor short-running queries, and a self-tuning mechanism periodically optimizes

scheduling parameters based on observed workload characteristics, without requiring any offline training.

The experimental evaluation demonstrates that the scheduler improves query performance for short-running queries compared to MonetDB by up to $4.5\times$ and compared to PostgreSQL by more than $65\times$. Although the design focusses on optimizing the execution of small queries, long-running queries also benefit with modest improvements in mean and maximum slowdown relative to other systems. Nevertheless, the scheduler is implemented within an existing DBMS and its performance improvements are therefore limited to this system.

2.4.1.2 Learned Scheduler

LSched (Learned Scheduler) [40] addresses the challenge of efficient query scheduling in in-memory block-based database systems. The proposed solution integrates a reinforcement learning-based scheduling predictor with a query encoder that extracts detailed operator, and query-level features. Similar to the integration of DuckDB into HARP, the approach utilizes application-specific insights to improve prediction accuracy.

Sabek et al. capture a selected set of transparent features from the physical query plans, including detailed work orders, as well as information about how these queries interact with the execution environment, such as the current state of active execution threads.

Based on this, LSched’s scheduling predictor is responsible for selecting which operators to start execution from, determining pipeline degree, and assigning a number of threads per query. The training of the predictor is conducted through reinforcement learning, where scheduling decisions are evaluated and reinforced based on actual query execution performance. Unlike HARP, this approach requires a dedicated training phase. However, it mitigates the associated cost by leveraging transfer learning, which enables the model to adapt to new workloads efficiently without starting from scratch.

Evaluated with several benchmarks, LSched outperforms state-of-the-art schedulers and heuristic approaches, improving average query duration by up to 35–50%.

2.4.2 NUMA awareness

In recent years, a substantial body of research has focused on improving performance in NUMA systems by optimizing the placement and migration of data and threads at runtime. The primary objective across these approaches is to minimize remote memory accesses, reduce contention on shared resources, and improve overall system throughput.

Data migration Several approaches focus on migrating data across NUMA nodes based on observed workload characteristics.

Psaroudakis et al. [37] propose an adaptive data placement algorithm for main-memory column-store DBMSs that monitors CPU and memory bandwidth utilization to dynamically redistribute data and balance load across sockets. By triggering table migration or repartitioning when imbalances occur and restricting inter-socket stealing for

memory-intensive tasks, their approach achieves up to 100 % throughput improvements compared to static placement policies.

A different strategy extends the `move_pages`¹ [29] system call with configurable migration policies, allowing main-memory databases to autonomously decide when and how to migrate pages based on workload intensity and migration costs [38]. By enabling partial migrations, optimizing synchronization modes, and tuning batch sizes, this approach achieves significant performance improvements, with reported throughput gains of up to 130 %.

Thread migration Another class of solutions focuses primarily on thread migration.

The work by Chiang et al. [7], for example, classifies tasks as memory-bound or compute-bound by analyzing memory access frequencies, subsequently migrating threads to suitable cores and adjusting processor frequencies accordingly. This approach not only achieves up to a 28 % performance improvement but also reduces energy consumption.

Most scheduling approaches implicitly assume symmetric interconnect topologies, where the cost of accessing memory across sockets is uniform. However, Gaud et al. [19] specifically address asymmetric interconnect topologies, where latencies vary significantly between socket pairs. By tracking memory accesses per instruction and adapting thread placement accordingly, their approach demonstrates that incorporating information about interconnect latencies significantly improves the likelihood of selecting an optimal scheduling decision.

Additionally, Lim et al. [27] monitor NUMA bus topology and runtime memory usage to guide thread placement, achieving throughput improvements of around 25 %.

Data and thread migration Finally, some solutions combine data and thread migration to holistically balance NUMA node utilization.

One method leverages memory controller bandwidth, interconnect traffic, and Last-Level Cache usage to inform placement decisions [21]. Threads with high contention are migrated to less contended NUMA nodes, improving cache locality and balancing memory traffic.

A different approach of Wu et al. [53] addresses performance challenges of in-memory file systems on NUMA multiprocessor systems, where thread and file placement, as well as memory congestion, significantly impact latency. The authors propose Hydras-RFCT, a two-layer design combining request-guided file migration and congestion-aware thread scheduling. Files are dynamically migrated based on historical request patterns to balance load, while threads are scheduled to minimize NUMA distance without overloading memory controllers or interconnects. Experimental results report throughput gains exceeding 112 %.

Inspiration for HARP The presented approaches demonstrate that careful thread placement combined with strategic data placement can significantly improve performance on NUMA systems. Most existing work in this area, however, focuses exclusively on

¹ `move_pages` transfers selected pages of a given process to designated memory nodes.

the NUMA problem, often building standalone systems that primarily aim at maximizing performance. In contrast, this thesis integrates NUMA awareness into HARP, a framework that is already designed not only to improve performance but also to reduce energy consumption. Building on prior insights, I extend HARP with NUMA-aware thread scheduling and data placement, thereby combining performance-oriented NUMA techniques with energy-aware resource management in a unified system.

3 DuckDB Scaling

In Section 2.1, I presented HARP as a resource management framework designed to improve application performance while reducing energy consumption on heterogeneous systems. This chapter presents the integration of the DuckDB DBMS as a new use case within HARP. The purpose of integrating DuckDB into HARP is to facilitate runtime-adaptive resource management to optimize database workloads. The subsequent sections describe the design considerations and implementation details underlying this integration.

3.1 Design

The section begins with an overview of the integration process, emphasizing the key objectives and challenges of incorporating DuckDB into HARP. Following this, I introduce the structure of a DuckDB program to give an overview of how DuckDB operates and can be interacted with. On this basis, I examine potential integration points within DuckDB that are suitable for interfacing with HARP.

3.1.1 Integration with HARP

I first recap the general procedure for integrating an application into the HARP framework and show how this applies to DuckDB. Building on this, I present two specific mechanisms offered by HARP that address the central challenges of integration.

3.1.1.1 Overview

As shown in Figure 2.1 in Section 2.1.2, the HARP framework manages applications including, for instance, OpenMP-based benchmarks such as `mg.C` and `ep.C`. The objective of this work is to also enable DuckDB applications to be managed within the same framework, thereby broadening the range of workloads that HARP can handle.

As outlined in Section 2.1.2, the central component of the HARP framework is the RM, which monitors and manages applications. To establish communication with the RM, DuckDB requires its own instance of `libharp`, which sets up the connection through the socket interface exposed by the RM. Once connected, the RM’s objective is to determine resource allocations for DuckDB that strike a balance between performance and energy efficiency. To guide these allocation decisions, the RM continuously monitors performance and energy consumption of the running DuckDB program. By default, the RM measures performance in IPS. However, the integration of DuckDB into HARP provides the opportunity to supply database-specific performance metrics. Such metrics are expected to offer a more precise reflection of DuckDB’s behavior than generic hardware-level

indicators and, consequently, to improve the quality of resource management decisions. It is therefore essential to extract and report relevant performance metrics from DuckDB.

Once the RM selects a resource allocation for DuckDB, this decision is communicated back to the database via `libharp`. Section 2.1.4 discusses that adjusting an application’s internal parallelism to HARP’s resource allocations is important for efficient resource utilization. Therefore, it is worthwhile to explore entry points in DuckDB that enable adapting its internal parallelism according to varying resource allocations.

In summary, integrating DuckDB into HARP entails addressing two key challenges. The first is enabling DuckDB to adapt its internal scaling mechanisms in accordance with HARP’s resource allocations. The second is providing database-specific performance feedback to guide the RM’s allocation decisions.

3.1.1.2 ScalableApplication feature

The foundation for adapting an application’s internal scaling is a dedicated interface provided by HARP. This interface, called **ScalableApplication**, extends the **MappingFeature** class introduced in Section 2.1.5. When the RM decides on a new resource allocation for a managed application, `libharp` invokes the `mapping_update()` function of the associated **MappingFeature** to inform the application about the updated mapping. Similarly, the **ScalableApplication** feature implements its own `mapping_update()` function. During registration of an application with the **ScalableApplication** feature, the developer needs to provide a callback function that will be executed whenever the RM chooses a new mapping. In this call, the application receives information about the number of cores allocated by the RM. This information can then be used to adjust the number of threads utilized by the program. For the integration of DuckDB into HARP, this requires the implementation of such a callback function, which is responsible for adapting DuckDB’s internal parallelism.

3.1.1.3 UtilityMeasure feature

To provide runtime performance metrics to HARP, a different interface is available that can also be leveraged for the integration of DuckDB. This interface is called **UtilityMeasure** and serves as the mechanism for reporting application-level performance information to the RM. From the perspective of a managed application, the central component of this interface is the `update_utility()` function. Whenever an application generates new performance information, it pushes the corresponding value via this function, and the data is subsequently stored within the **UtilityMeasure**. When the RM requires updated performance metrics, it retrieves the most recent performance value via the **UtilityMeasure** feature. The retrieval of these performance values is entirely managed by the RM and remains decoupled from the application. Consequently, from the application’s perspective, the sole responsibility is to provide updated performance data, while the RM decides when and how to make use of it. For the integration of DuckDB, this implies that the system needs to be extended to provide appropriate database-specific performance metrics that can be exposed through this interface.

3.1.2 Identifying integration points

After establishing in the previous section how applications can interface with the HARP framework, the focus now shifts to DuckDB. First, I provide an overview of a representative DuckDB program to establish the foundation for the subsequent integration steps. Building on this, I continue by identifying suitable entry points within DuckDB's internal parallelism that can be adapted to reflect resource allocations communicated through the `ScalableApplication` feature. Finally, I determine how database-specific performance metrics can be extracted to provide them to HARP via the `UtilityMeasure` interface.

3.1.2.1 Structure of a DuckDB program

```
#include "duckdb.hpp"

int main() {
    DuckDB db(nullptr);
    Connection con(db);
    con.Query("CREATE TABLE integers (i INTEGER, j INTEGER)");
    con.Query("INSERT INTO integers VALUES (3, 4), (5, 6)");
    auto result = con.Query("SELECT * FROM integers WHERE i = 3");
    result->Print();
    return 0;
}
```

Listing 3.1: Example for a simple DuckDB program

A basic example of using the DuckDB C++ API is illustrated in Listing 3.1. To interact with DuckDB, the program first initializes a `DuckDB` instance, which sets up the underlying database. Subsequently, a connection to this database is established via the `Connection` class. Once created, this connection can be used to issue queries to the database and retrieve the corresponding results. As demonstrated in the example, these queries can serve to create tables, populate them with data, and subsequently request specific information.

Instances of the `DuckDB` and `Connection` classes constitute the minimal setup necessary for interacting with the DuckDB DBMS, and are therefore assumed to be present in essentially every DuckDB program [14]. Their central role makes them particularly relevant for further investigation, as they provide potential entry points for adapting DuckDB's internal scaling mechanisms and for collecting database-specific performance information.

3.1.2.2 DuckDB class

The `DuckDB` class serves as the core interface for database initialization and encapsulates key structural components of the database system. It maintains the catalog, which is a repository of metadata containing information about the database's objects, such as tables, views, indexes, and their associated schemas [15]. In addition, the class owns the

primary database instance, which encompasses various subsystem managers responsible for connections, buffer handling, and storage management. Notably, it also incorporates an instance of a **TaskScheduler** class.

When a query is executed, it is decomposed into smaller units called *tasks*, which can run in parallel and correspond to database operators such as table scans or hash joins. Scheduling in DuckDB operates at the task level, with the **TaskScheduler** responsible for assigning tasks to threads and managing their execution. Since the **TaskScheduler** controls thread creation and execution, it presents a promising entry point for adapting DuckDB’s internal parallelism in accordance with the resource allocations provided by HARP, enabling a more fine-grained and efficient utilization of available compute resources.

3.1.2.3 Connection class

The previous section outlined that the **DuckDB** class holds metadata and provides access to the **TaskScheduler**, enabling thread adaptation based on HARP’s resource assignments. However, for effective resource management, it is also essential to obtain performance data from DuckDB to assess the impact of different resource configurations on query execution. Therefore, the following discussion examines the **Connection** class, which provides the primary interface for interacting with the database and orchestrating query execution.

Within each **Connection**, a **ClientContext** encapsulates session-specific information and manages the execution state for that particular **Connection** [15]. Furthermore, the **ClientContext** class is responsible for the actual execution of queries in DuckDB. When a query is issued through a **Connection**, as in Listing 3.1, it is directly forwarded to the corresponding **ClientContext**. Therefore, it could be possible to track when a query starts and finishes through the **ClientContext**. The runtime of a query may serve as a useful indicator for assessing performance, making the **ClientContext** a promising entry point for providing HARP with database-specific performance information.

In addition, the **ClientContext** allows the configuration of session-local settings, such as enabling a **ProgressBar** and a **QueryProfiler**. The **ProgressBar** tracks the number of rows processed during query execution, while the **QueryProfiler** records which operators are employed throughout the execution. Both mechanisms provide the potential to obtain more fine-grained performance metrics than those derived solely from query-level measurements.

3.2 Implementation

The previous section identified the **DuckDB** and **Connection** classes as central entry points for interfacing with DuckDB. Building on this foundation, the current section details the concrete implementation of the integration into HARP. It explains how the identified classes are utilized in the communication between DuckDB and HARP, how resource adaptation is triggered and how relevant runtime information is extracted.

3.2.1 Scaling with DuckDB

To achieve runtime adaptation of DuckDB’s parallel execution, the `TaskScheduler` class has been identified as a promising entry point, as examined in Section 3.1.2.2. The following section provides a detailed description of how the `TaskScheduler` can be leveraged to implement dynamic adjustments of DuckDB’s internal parallelism.

3.2.1.1 Adaptation through DuckDB settings

After identifying the `TaskScheduler` as a promising entry point for adapting DuckDB’s internal parallelism, the next step is to determine how its thread management can be modified so that the number of threads aligns with resource allocations provided by HARP. DuckDB provides an internal configuration interface, the *DuckDB settings*, that exposes a wide range of runtime parameters, including memory limits, logging options, and — most importantly — the number of threads available to the `TaskScheduler`. While this interface at first appears suitable for dynamic adaptation, practical experimentation reveals two significant limitations.

First, DuckDB differentiates between several types of threads, yet the settings interface only allows adjusting one specific thread type, whereas a mechanism to directly set the number of all threads would be more desirable. Second, updating the thread count via the configuration interface does not immediately reconfigure the underlying thread pool. Instead, the `TaskScheduler` only applies changes in thread counts when its `RelaunchThreads()` method is invoked, which occurs only after the completion of the current query.

This deferred realization of thread updates poses a challenge in scenarios where HARP modifies the resource allocation of a DuckDB application during query execution. In such cases, the internal parallelism cannot be adapted until the running query terminates, leaving the system in a state where, for potentially long periods, the query executes with a mismatched number of threads relative to the available cores. While this limitation may be tolerable for short queries, it is more problematic for long-running workloads, where such a mismatch can significantly degrade performance.

In situations where no pre-generated OPs are provided to the RM, as explained in Section 2.1.2 (8), the issue is further exacerbated. At the beginning of application execution, the RM frequently changes configurations to explore how the application performs and consumes energy under different resource allocations. If such adjustments occur while a long-running query is active, DuckDB continues operating with an outdated thread configuration, potentially leading to worse performance than would be observed under immediate adaptation. This, in turn, may mislead the RM into drawing inaccurate conclusions about the application’s behavior under specific configurations.

Since a robust solution must be applicable not only to short queries but also to long-running workloads, relying on DuckDB’s existing configuration interface proves insufficient. Consequently, an alternative approach to adapting the internal parallelism is required.

3.2.1.2 Directly adapting the TaskScheduler class

To overcome the limitations encountered when using DuckDB’s settings interface, I explored a lower-level approach by directly interfacing with the `TaskScheduler` instance. To determine the required steps, I start with analyzing how the settings interface internally performs thread reconfiguration. It essentially carries out two operations. First, it invokes the `SetThreads()` method of the `TaskScheduler`, which updates the scheduler’s internal thread counts. Second, it modifies a global configuration object that maintains, among other parameters, the current thread count.

By calling the `SetThreads()` method directly, it becomes possible to adjust the total number of threads across all thread types, rather than being limited to the subset allowed by the standard interface. Following this, the `RelaunchThreads()` method can be immediately invoked to trigger the scheduler’s reconfiguration process.

As outlined in Section 3.1.1.2, integrating this approach with HARP requires leveraging the `ScalableApplication` feature, which enables the registration of a callback function that is executed whenever the RM decides on a new resource allocation. First, the DuckDB instance must register for this feature through its `libharp` interface, ensuring that the registered callback function is invoked whenever the resource allocation changes.

```
scale_duckdb(nr_threads):
    task_scheduler = duckdb.GetScheduler()
    task_scheduler.SetThreads(nr_threads)
    duckdb.config.thread_count = nr_threads
    task_scheduler.RelaunchThreads()
```

Listing 3.2: Pseudo code for adapting DuckDB’s internal parallelism through the `ScalableApplication` feature

Within this callback function, the new approach is applied as outlined in Listing 3.2. First, the `SetThreads()` method of the `TaskScheduler` is called to set the total thread count to match the number of cores assigned by the RM. As discussed in Section 3.1.2.2, the `DuckDB` class encapsulates the active `TaskScheduler` instance. To access it, I implemented a function within `libharp` that retrieves a reference to the currently active `DuckDB` object and, consequently, its `TaskScheduler`. The constructor of the `DuckDB` class is modified to invoke this function at the end of its initialization process. Since the `DuckDB` object is constructed prior to the execution of any queries, this ensures that the internal parallelism can be configured before meaningful work begins, which is crucial for maintaining adaptation consistency.

Once access to the `TaskScheduler` is established, the global configuration object can be updated to reflect the new thread count, again accessed via the `DuckDB` object. Finally, the `RelaunchThreads()` method is invoked to initiate the reconfiguration of the thread pool, completing the dynamic adaptation of DuckDB’s internal parallelism in response to HARP’s runtime resource allocations.

3.2.1.3 Reconfiguration limitation

Until now, it was assumed that directly invoking the `RelaunchThreads()` method of the `TaskScheduler` would immediately reconfigure the threads. However, closer examination reveals that this is not the case. Analyzing the implementation of the `RelaunchThreads()` method reveals the underlying reason. The method first checks the current number of active threads against the target thread count set via the `SetThreads()` function. If the number of running threads exceeds the target, the method joins all threads before starting the required number of new threads. However, this join operation blocks until the currently executing query finishes, effectively deferring the realization of the thread updates, similar to the approach discussed in Section 3.2.1.1. Conversely, if the number of active threads is below the target, new threads are spawned immediately but remain idle until the next query begins.

As a result, although directly modifying the `TaskScheduler` allows for a more precise specification of thread counts, DuckDB’s internal task scheduling model defers workload distribution to query boundaries. This behavior limits the granularity and timeliness of dynamic adaptations in response to changes in resource allocation. At this point, this suboptimal method remains the practical choice, as implementing an immediate adaptation mechanism would likely require a comprehensive restructuring of DuckDB’s thread and task management, which is beyond the scope of this work.

3.2.2 Providing feedback to the HARP RM

In the previous section, I discussed two approaches to adapt DuckDB’s internal scaling mechanisms in response to HARP’s resource allocation decisions. The other key aspect of integrating DuckDB into HARP involves providing application-specific performance feedback to the RM, which is the focus of this section.

As outlined in Section 3.1.2.3, I identified three potential methods to obtain performance metrics at different levels of granularity. The following discussion systematically examines these methods, evaluating their suitability for providing HARP with accurate and meaningful performance information.

3.2.2.1 Queries

As discussed in Section 3.1.2.3, each `Connection` in DuckDB is associated with a `ClientContext`, which is responsible for issuing queries. This makes the `ClientContext` a robust entry point for tracking query lifecycles. Although DuckDB exposes multiple methods for submitting queries, all of them eventually invoke a central function within the `ClientContext` where query execution formally begins. This provides a suitable location to insert performance tracking.

As outlined in Section 3.1.1.3, HARP provides the `UtilityMeasure` interface as a common mechanism through which applications can push performance metrics, while the RM independently pulls and evaluates them. Similar to the `ScalableApplication` feature, applications must register their use of the `UtilityMeasure` through `libharp`, thereby signaling to the RM that application-specific metrics are available, ensuring that the generic IPS-based metric is not used instead.

To realize this integration for DuckDB, I implement a function within DuckDB's `libharp` instance that calls the `update_utility()` function of the `UtilityMeasure`. Subsequently, I modify the `ClientContext` function responsible for initiating query execution to measure the elapsed query time. Once the query completes, this measured time is passed to the function in DuckDB's `libharp` instance, which is responsible to invoke `update_utility()`. Since HARP interprets higher utility values as indicators of better performance, the raw execution time needs to be transformed into a suitable metric. The most natural choice, and the one adopted here, is to derive a queries-per-second metric, which is pushed to the `UtilityMeasure` feature upon every query completion.

While this mechanism provides a valid performance measure, it faces limitations in scenarios with long-running queries. In such cases, metrics are only reported at query boundaries, potentially resulting in infrequent updates that fail to capture the application's short-term performance behavior. This poses a challenge for HARP, which may require more fine-grained and frequent updates to effectively evaluate how the application responds to different resource allocations. For this reason, it is worthwhile to also investigate mechanisms that enable more detailed performance insights beyond query-level measurements.

3.2.2.2 Operators

As described in Section 3.1.2.3, DuckDB provides the possibility to enable a `QueryProfiler` through the `ClientContext`, allowing profiling at the level of individual operators during query execution. By default, the `QueryProfiler` is deactivated and must be explicitly enabled for the current session. Since this activation is performed via the `ClientContext` of a `Connection`, a mechanism similar to the one introduced in Section 3.2.1.2 is required, where the constructor of the `Connection` class is modified so that the `QueryProfiler` is enabled upon initialization. This activation itself can be realized through DuckDB's settings interface, introduced in Section 3.2.1.1.

Once the profiler is active, the central problem lies in making practical use of the information. DuckDB's profiling infrastructure exposes two key functions for that. One is invoked when an operator begins execution, and the other when it completes. Since operators are executed frequently, it would be inefficient to report profiling data after each invocation. Instead, I extend the profiling structure so that it invokes a function in DuckDB's `libharp` instance, which only increments a counter whenever an operator completes. This counter is periodically read by a dedicated monitoring thread. From this data, I derive an operators-per-second metric, which can then be pushed to the `UtilityMeasure` feature of HARP.

While it is possible to maintain a single counter aggregating all operator types — e.g., table scans, hash joins, projections — it is equally feasible to track separate counters for different operators. The latter option may offer more expressiveness, as, for instance, a high rate of hash joins could indicate stronger performance characteristics than an equivalent rate of table scans. However, introducing multiple counters raises the challenge of condensing them into a single derived metric, as HARP is only able to process a unified performance value.

Given that a single aggregated counter may not capture sufficient detail, while maintaining multiple operator-specific counters requires prior knowledge about their relevance and an appropriate combination strategy, operator-level profiling may not be the most suitable solution. For this reason, it is valuable to further explore alternative sources of performance metrics.

3.2.2.3 Rows

In addition to the `QueryProfiler`, DuckDB also provides the option to activate a `ProgressBar`. The `ProgressBar` class tracks query execution progress, specifically the number of rows that have been processed by the current query. Internally, the `ProgressBar` maintains an update function that is invoked periodically to refresh progress information. To make use of this, I adapt this update function such that the refreshed progress data is handed over to a function within DuckDB's `libharp` instance. In this function, I combine the reported number of processed rows with the current timestamp to derive a rows-per-second metric, which is subsequently pushed to the `UtilityMeasure` feature whenever the progress bar updates.

Compared to the previously discussed queries-per-second metric, the rows-per-second approach offers finer granularity, as performance information is provided continuously during query execution rather than only upon query completion. Furthermore, unlike the operators-per-second metric, it does not require prior knowledge about which operator types are most relevant or how to meaningfully combine them into a single representative metric. For these reasons, this work adopts rows-per-second as the preferred performance metric for integration with HARP.

4 NUMA Awareness

As outlined in Section 2.3.3, failing to consider NUMA topology during scheduling can lead to significant performance degradation. To address this challenge, the second part of this work focuses on extending HARP with support for NUMA architectures, enabling scheduling decisions that account for memory locality and aim to reduce latency-related performance penalties. This chapter examines the principal design considerations and implementation strategies required to make HARP NUMA-aware.

4.1 Design

Before detailing the implementation, it is necessary to consider the different approaches for integrating NUMA-related information into HARP. Understanding the possible methods and trade-offs provides a foundation for implementing the mechanisms effectively. I structure the design in three main parts. First, I explain how NUMA-related information can be made accessible to HARP, which is essential to reason about memory locality and core placement. Second, I describe how this information can be incorporated into resource allocation decisions, allowing HARP to schedule threads in a way that reduces memory latency and contention. Finally, I present an approach that allows applications to adapt to thread migrations across NUMA nodes by re-establishing memory locality and thereby maintaining efficient application performance.

4.1.1 NUMA topology data

Before HARP can make NUMA-aware scheduling decisions, it first needs a clear picture of the system's NUMA layout. This section discusses which NUMA topology information HARP requires and how to collect this data and make it accessible for HARP.

4.1.1.1 Relevant NUMA information

As explained in Section 2.3, memory access latency depends heavily on the underlying NUMA topology. Typically, accessing memory within the same NUMA node is the fastest. Accessing memory across nodes on the same socket incurs higher latency, while accesses across different sockets are even slower, as illustrated in Figure 2.3. To account for these differences, HARP distinguishes between these three levels of memory access. To do this effectively, it first needs to know how many sockets exist in the system. Then, it must determine how many NUMA nodes exist and which nodes belong to which socket. Similarly, it needs to know which CPU cores belong to which NUMA node. Finally, HARP requires information about the NUMA distances between nodes to assess the scale

of access latency differences between nodes. Together, this set of information should be sufficient to enable NUMA-aware scheduling decisions within HARP.

4.1.1.2 Gathering the information

A first approach to integrating hardware topology information into HARP is to utilize data exposed through the virtual file system `sysfs`¹ [32] on Linux. This interface allows enumeration of available NUMA nodes, the CPU cores assigned to each node, as well as the distances between nodes. Similarly, information about the socket affiliation of each core can also be obtained via `sysfs`.² While this method provides HARP with a direct and standardized way of accessing the system topology, it primarily reflects the static hardware layout and offers limited flexibility for experimenting with modified or abstracted topologies.

```
core_types:
- {type: "P-core", threads: 2, power_coefficient: 1}

numa_nodes:
- {id: 0, distances: [10, 12, 12, 12, 21, 21, 21, 21]}
- {id: 1, distances: [12, 10, 12, 12, 21, 21, 21, 21]}
- {id: 2, distances: [12, 12, 10, 12, 21, 21, 21, 21]}
- {id: 3, distances: [12, 12, 12, 10, 21, 21, 21, 21]}
- {id: 4, distances: [21, 21, 21, 21, 10, 12, 12, 12]}
- {id: 5, distances: [21, 21, 21, 21, 12, 10, 12, 12]}
- {id: 6, distances: [21, 21, 21, 21, 12, 12, 10, 12]}
- {id: 7, distances: [21, 21, 21, 21, 12, 12, 12, 10]}

sockets:
- {id: 0, NUMA_nodes: [0, 1, 2, 3]}
- {id: 1, NUMA_nodes: [4, 5, 6, 7]}

equivalence_scheme: "core-type"
energy_measure_method: "perf"
```

Listing 4.1: Example hardware description file extension for a system with two Intel Xeon Max 9468 CPUs

An alternative approach is to extend the existing hardware description file of HARP, which was introduced in Listing 2.1 in Section 2.1.2. The proposed adaptations are illustrated in Listing 4.1. This strategy integrates naturally with the framework’s established mechanism for supplying hardware characteristics and is the preferred approach in this thesis.

¹ Concretely, this information is available at the path `/sys/devices/system/node/`.

² It can be accessed at the path `/sys/devices/system/cpu/cpu*/topology/physical_package_id`.

4.1.2 Integrating NUMA information into HARP’s scheduling decisions

Following the discussion on making NUMA-related information accessible to HARP, it becomes essential to examine how this information can be incorporated into HARP’s scheduling decisions. Up to this point, the HARP RM — in the following also referred to as *server* — already determines which applications benefit from high-performance P-cores and which are more suitable for E-cores, aiming to balance system-wide performance and energy efficiency. Integrating NUMA awareness into scheduling decisions requires identifying which components of the mapping process need to incorporate topology information. The process of determining a new mapping for a client can be conceptually divided into two main stages, as explained in Section 2.1.3. The first stage is the LR-based approximation algorithm, which determines the number of cores of each type assigned to a client. Depending on the selection of cores, certain resource allocations could be realized NUMA-local, across NUMA nodes but remaining socket-local, or fully cross-socket. For example, referring back to the system illustrated in Figure 2.3 in Section 2.3.2.2, if the algorithm allocates four cores to an application, several distributions are possible. All four cores could reside within a single NUMA node, two cores could be placed across two NUMA nodes within the same socket, or the cores could be distributed across all four NUMA nodes. For applications expected to suffer from poor memory locality, a NUMA-local allocation should be preferred here to preserve performance. The second stage is the concrete assignment of applications to specific cores. Here, knowledge of the NUMA topology is again crucial: applications for which NUMA-local placement is desired should be assigned to cores residing on the same node.

Integrating NUMA information at both stages provides the possibility for the server to enhance memory locality while maintaining an efficient balance of performance and energy consumption across the system.

4.1.3 Approximating performance loss

When NUMA topology is considered, additionally questions arise regarding the memory-intensity of applications: certain workloads may rely heavily on memory access and therefore benefit more from maintaining good memory locality, whereas compute-heavy applications may be less sensitive to NUMA placement. Consequently, it becomes important to find a way for the server to assess the memory intensity of applications in order to guide scheduling decisions. This can be achieved either by having the client provide memory-intensity information directly or by allowing the RM to infer it through runtime observation.

Getting memory-intensity from the client When a client connects to the RM, it can provide pre-generated OPs via a mapping file, as described in Section 2.1.2 (5). In principle, this mechanism could be extended to allow the client to also supply information regarding its sensitivity to NUMA placement. This approach, however, relies on having performance measurements available beforehand, which is generally unsuitable for general-purpose desktop systems, where applications are often unknown a priori and neither users nor developers are expected to conduct such measurements.

Determining memory-intensity on the server-side To remain consistent with HARP’s design goal of managing applications dynamically without the need for prior knowledge, enabling the server to evaluate an application’s potential sensitivity to poor memory locality is preferable. However, the server typically has no knowledge of future client behavior, particularly regarding where and how frequently memory will be accessed. To make informed scheduling decisions, it must therefore rely on assumptions to estimate the potential performance impact when migrating an application from one NUMA node to another.

One possible indicator for estimating such effects is the NUMA distance, which quantifies the relative performance degradation associated with accessing remote memory compared to local memory. However, I expect purely relying on NUMA distances to be insufficient, since presumably not every instruction executed by an application involves memory access. For many workloads, only a small fraction of executed instructions actually accesses memory. In the following, this ratio between memory-related and total instructions is referred to as the *memory heaviness* of an application.

My proposal is to combine NUMA distances and memory heaviness into a unified formula, enabling the calculation of a performance degradation factor. This factor could then be used to estimate the expected performance loss when migrating a particular application from its current NUMA node to another, thereby supporting more informed and NUMA-aware scheduling decisions within HARP.

4.1.4 Thread migration across NUMA nodes

When an application is migrated from one NUMA node to another, it will typically already have allocated its memory on the original node where it was first executed. This tight coupling of execution threads and memory placement gives rise to a fundamental challenge: if the threads are moved, but the memory remains on the original node, all subsequent accesses to this memory are performed remotely, leading to persistently higher latencies. Such remote accesses can be particularly detrimental for memory-intensive applications, which may suffer significant slowdowns under this configuration. An alternative strategy is to migrate the application’s memory pages together with its threads to the new node. While this approach introduces a one-time overhead for transferring the data, it restores locality once the transfer is complete and thereby enables low-latency memory accesses during subsequent execution. In essence, the trade-off lies between paying a constant penalty due to remote memory access or incurring a one-time cost in exchange for improved locality thereafter.

The decision of whether to move memory alongside threads can be realized either at the server level, where decisions are made with a global view of all applications, or at the client level, where the application itself takes part in managing memory placement. The following section discusses these two perspectives in detail.

4.1.4.1 Server-side memory placement

One possible approach is to delegate decisions about memory placement to the server. In this design, the server would not only determine the migration of application threads

across NUMA nodes but also manage the relocation of the corresponding memory regions. In principle, the server is well-positioned to decide *where* memory should be moved, since it initiates the thread migrations and therefore has precise knowledge of the source and target NUMA nodes.

However, determining *when* it is worthwhile to transfer memory is substantially more challenging. The server has no direct insight into the future behavior of its managed applications, and can therefore only rely on heuristic assumptions derived from previous memory access patterns. Additionally, applications may implement their own memory management policies, which could be violated if the server unilaterally decides to move memory. This uncertainty complicates the decision-making process, as the server must balance the one-time overhead of migrating memory against the potential benefits of improved locality while respecting application-specific requirements.

Moreover, the server typically manages multiple applications simultaneously. Extending its responsibilities from thread scheduling to memory placement would significantly increase its workload and introduce additional runtime overhead, thereby reducing the efficiency of the scheduling process. Taken together, these considerations suggest that server-side memory management is not an ideal solution. Consequently, it is worth exploring whether memory placement can be handled more effectively at the client level, where application-specific knowledge is available.

4.1.4.2 Client-side memory placement

An alternative approach is to delegate memory placement decisions to the client. Unlike the server, the client has detailed knowledge of its own workload characteristics, including the frequency and locality of future memory accesses. This information enables the client to make more informed decisions about whether it is beneficial to accept the overhead of migrating memory when being moved to a new NUMA node.

To realize this, one possible method for the client is to implement a `MappingFeature`, introduced in Section 2.1.5, which notifies the client about changes in thread placement. When a change does involve moving threads to a different NUMA node, the client could then decide whether to retain its memory on the original node, accepting the cost of remote memory accesses with higher latencies, or to migrate the data to the new node, paying a one-time transfer cost but potentially benefiting from improved locality thereafter.

Memory placement does not necessarily require knowledge about the behavior or resource demands of other applications. Therefore, each application can independently determine its own strategy for handling memory migrations based solely on its internal workload properties. This makes client-side decision-making particularly suitable in this context, as it avoids overloading the server with additional responsibilities while still enabling applications to exploit NUMA-aware optimizations effectively.

4.2 Implementation

In the previous section, I analyzed and compared different strategies for making NUMA information accessible, incorporating NUMA awareness and memory heaviness into

scheduling decisions, and handling memory placement when having thread migrations across NUMA nodes. Building on this foundation, the following section presents the concrete implementation of these mechanisms within HARP.

4.2.1 Resource allocation adaptations

Following the design outlined in Section 4.1.2, this section details how NUMA awareness is realized within HARP. Implementation tasks include enabling the server to assess the memory heaviness of applications, incorporate topology information during resource allocation, and assign applications to specific cores in a NUMA-aware manner. The following subsections describe how these design principles are translated into concrete mechanisms within HARP.

4.2.1.1 Adaptations of OP measurements

As outlined in Section 2.1.2, OPs are the central building blocks of the resource allocation process, as they capture the performance and energy characteristics of applications under specific configurations. During online measurements, HARP attempts to capture application behavior dynamically by collecting OPs for different configurations at runtime, which provides the basis for informed scheduling decisions. Therefore, it is essential to ensure that OPs accurately represent the relevant properties of the system. In the context of enabling NUMA awareness, this requires revisiting both the structure of OPs and the way their measurements are obtained and interpreted. Adapting OPs accordingly establishes the foundation upon which NUMA-aware resource allocation strategies can be built.

Extending the OP structure When the server determines a resource allocation for an application, it does so by selecting an OP, which represents a concrete application configuration. During execution of the application at this configuration, the server typically measures performance and energy consumption to evaluate how well the configuration performs. These measurements are subsequently used within the RAA, introduced in Section 2.1.3, to guide the selection of future configurations. Each OP captures information about the number of cores of each type and the number of threads assigned, but it does not specify which exact cores are used, as presented in Listing 2.2 in Section 2.1.2. However, as discussed in Section 4.1.2, some resource allocations could be realized with a NUMA-local, socket-local and cross-socket topology level. Given that the NUMA topology of a system can substantially affect application performance, it is essential to capture this influence within the measurement and evaluation process. To this end, the OP structure is extended to include information about the topology level at which a configuration is executed. As proposed in Section 4.1.1.1, it is expected to be sufficient to distinguish between configurations that run entirely on a single NUMA node, those that span multiple nodes within the same socket, and those that execute across different sockets. Incorporating this level of topology provides the potential to more accurately correlate resource configurations with observed performance and energy characteristics, enabling more informed allocation decisions.

Comparability of OPs When evaluating different OPs within the RAA, it is not sufficient to consider only the topology level under which the performance was measured. It is equally important to ensure that OPs are comparable. For instance, two configurations that differ slightly in terms of core and thread counts may exhibit substantially different measured performance if one was executed across sockets while the other was executed within a single NUMA node. In such cases, the NUMA-local measured configuration could appear superior and be preferred by the RAA, even though the other configuration might achieve better performance if it were also executed NUMA-local.

```
if (op.numa_level == CROSS_SOCKET && op.optimal_level == NUMA_LOCAL)
{
    op.numa_level = op.optimal_level;
    op.utility *= 1 + _cross_socket_penalty;
}
```

Listing 4.2: Adapting Operating Point measurements

To improve comparability, I chose not to record OP measurements strictly based on the topology level actually used during execution. Instead, I use the optimal level at which the configuration could theoretically run given the assigned cores. This approach is presented in Listing 4.2. For instance, if a configuration was executed across multiple sockets but could have been confined to a single NUMA node, its OP measurement is recorded as NUMA-local. Additionally, the utility of the OP needs to be slightly adjusted to reflect the potential improvement achievable under this better topology. This adapted recording of OP measurements enables a fairer comparison among configurations.

4.2.1.2 Adapting the LR Mapper

As discussed in Section 4.1.2, the first stage of the mapping process that requires NUMA awareness is the optimization step, in which an OP is selected for each application. In HARP, the optimization problem is solved using an LR-based approximation algorithm, which identifies OPs that strike a balance between performance and energy consumption. The fundamental principles of this algorithm were introduced in Section 2.1.3. To extend this approach, the algorithm needs to be adapted so that it not only decides how many threads and cores each application should receive, but also at which topology level these resources should be executed. This extension ensures that the NUMA topology, which can significantly influence performance, is systematically incorporated into the decision process.

Consideration of worse topology levels The adaptation is realized in three main steps and is illustrated in Listing 4.3.

First, the algorithm proceeds as before, selecting the OP that best balances performance and energy consumption. To recap from Section 4.2.1.1, for comparability, OPs are assumed to contain measurements for their best possible topology level. For example, consider the system presented in Figure 2.2 in Section 2.3.2.2, comprising two sockets, each containing two NUMA nodes with six cores per node. If an application *A* requests

```

for client in clients:
    best_op = find_best_op(client.ops)

    numa_local_op    = MAX_VAL
    socket_local_op  = MAX_VAL
    cross_socket_op  = MAX_VAL

    match best_op.numa_level with:
        CROSS_SOCKET:
            cross_socket_op = best_op
        SOCKET_LOCAL:
            socket_local_op = best_op
            cross_socket_op = estimate_op(best_op, CROSS_SOCKET)
        NUMA_LOCAL:
            numa_local_op    = best_op
            socket_local_op  = estimate_op(best_op, SOCKET_LOCAL)
            cross_socket_op  = estimate_op(best_op, CROSS_SOCKET)

    return min(numa_local_op, socket_local_op , cross_socket_op)

```

Listing 4.3: Pseudo code for adapting the LR-based approximation algorithm

eight cores, the best achievable topology level for this configuration is socket-local, since no single NUMA node contains eight cores. This socket-local configuration serves as the reference.

Second, the algorithm creates a new OP with the same configuration but at a less favorable topology level, in this case cross-socket, and with degraded performance, which is particularly pronounced for applications exhibiting high memory heaviness. This second step allows the algorithm to evaluate the trade-off between performance and energy efficiency for both the optimal and suboptimal topology levels of a given OP. In the presented code, a smaller value corresponds to a better trade-off.

Third, the OP exhibiting the best performance-energy trade-off is selected and returned. The remainder of the algorithm follows the original procedure: the evaluation is repeated for each client, and the algorithm verifies whether the total number of required cores of each type exceeds the available hardware resources. If the resource demand is within the hardware limits, a valid solution is obtained and the approximation algorithm terminates. Otherwise, the algorithm is rerun until a feasible allocation is determined.

Incorporating a NUMA-penalty If the algorithm were left at this stage, it would almost never select an OP with a worse topology level, as this would yield lower performance without reducing energy consumption, thereby producing a less favorable performance-energy trade-off. However, good memory locality is a limited resource: when multiple applications simultaneously request NUMA-local or socket-local execution, these demands cannot always be satisfied. For instance, on the same example system illustrated in Figure 2.2, if three applications each require eight cores executing socket-local, only two of them can be executed socket-local, and the third must be

Greedy search While the RAA determines how many threads an application should run, how many cores of each type it should receive, and at which topology level the application should execute, it remains the task of the instantiation process to translate these high-level decisions into concrete thread-to-core mappings that respect the NUMA-aware allocation results. To perform this mapping, several inputs are required. From the RAA, we obtain for each application the number of cores of each type, as well as the desired topology level. The hardware description file specifies which CPU cores belong to which NUMA node and which NUMA nodes are located on which socket, as presented in Listing 4.1. Additionally, the algorithm maintains information about currently available cores. Based on this information, the instantiation process performs a greedy search to assign the requested cores to each application. If an application is to be scheduled NUMA-local, the algorithm checks each NUMA node sequentially to find one with sufficient free cores of the required types and assigns them if possible. For socket-local applications, the process works analogously at the socket level, selecting cores within a single socket when available. Finally, for applications marked as cross-socket, there are no locality constraints, so the algorithm assigns the first free cores it encounters.

Ordering of applications However, this greedy strategy introduces a potential pitfall. Without additional precautions, an application permitted to run cross-socket could inadvertently consume free cores on a NUMA node or socket needed by another application requiring stricter locality guarantees. To prevent this, I introduce a priority-based ordering of applications during core assignment. Applications that shall run NUMA-local are processed first, followed by those that require socket-local execution, and finally those without topology constraints. This ordering ensures that applications benefiting most from strong locality receive their cores before less constrained workloads.

Overlapping requirements A further challenge arises when the topology-level requirements of all applications cannot be satisfied simultaneously. Without considering NUMA awareness, HARP already provides mechanisms to handle infeasible allocations by relaxing constraints and allowing co-allocation of cores between different applications. With NUMA awareness integrated, however, I must also adapt this process to account for cases where locality constraints cannot be fully respected. As described in Section 4.2.1.2, when an infeasible situation is detected, the RAA is re-executed with penalties applied to highly demanded topology levels. These penalties are incrementally increased in subsequent iterations. Eventually, the penalty becomes large enough that no application benefits from selecting a better topology level, causing the system to naturally fall back to a configuration equivalent to cross-socket execution. From this point onward, the allocation proceeds as in the original HARP mechanism, where overlaps in core usage are handled explicitly.

4.2.2 Approximating performance loss

In the previous section, I discussed how the OP structure and the RAA can be adapted to incorporate NUMA awareness. To ensure comparability of operating points, as described in Section 4.2.1.1, and to account for different topology levels within the LR-based

approximation algorithm, outlined in Section 4.2.1.2, it becomes necessary to estimate how performance and energy metrics observed at one topology level — e.g., NUMA-local — would change if the same configuration were executed at a different topology level — e.g., cross-socket.

In this section, I examine the approach proposed in Section 4.1.3, where the goal is to construct a formula that approximates the potential performance degradation or improvement when migrating an application from one NUMA node to another.

4.2.2.1 Obtaining Core Elements

The first component of the model is based on NUMA distances, which are already available within the hardware file and can therefore be directly used. To represent the metric of memory heaviness, which was introduced in Section 4.1.3, I utilize a ratio of Last-Level Cache (LLC) misses to the total number of executed instructions. Since these performance metrics are already collected within the HARP framework, memory heaviness can be integrated without introducing additional measurement overhead.

4.2.2.2 Proposed Model

To combine NUMA distances and memory heaviness into a unified formula, I propose the following power-law model:

$$f(d_0, d, m) = \alpha \cdot m^\beta \cdot (d - d_0)^\gamma$$

where d_0 denotes the current topology level used as reference, d is the target topology level, m represents the memory heaviness of the application, α is a linear scaling factor, β is an exponent modeling the impact of memory heaviness and γ is an exponent modeling the impact of topology distance.

This model is intentionally flexible as by adjusting β and γ , it can represent linear, quadratic, cubic, root, or logarithmic relationships between performance degradation, NUMA distance, and memory heaviness. Given the absence of prior knowledge about the exact influence of these factors, such flexibility is essential.

While d_0 , d and m are given or measurable, the parameters α , β and γ have to be determined empirically. To achieve this, I generate and evaluate a set of controlled test cases. Afterward, I analyze the resulting data using linear regression in logarithmic space, which is assumed to provide a relatively stable estimation while requiring only a limited number of samples [51]. The goal is to identify a formula that, at least approximately, fits a representative range of applications.

4.2.2.3 Experimental Setup

To evaluate the proposed approach, I conduct experiments on a dual-socket server equipped with two Intel Xeon Max 9468 CPUs. I select three types of applications with different degrees of memory heaviness. The NPB [3], specifically the Embarrassingly Parallel (EP) benchmark, represent applications expected to show minimal sensitivity to memory locality. Furthermore, I develop a custom Random Access (RA) program as

Application	Memory Heaviness (LLC misses / 10^3 Instr.)	Performance Loss in %	
		Intra-Socket	Cross-Socket
ep.C	0.01	-0.6	-0.1
DuckDB query	2.00	1.7	10.5
RA(0)	35.00	4.4	55.5
RA(40)	6.50	17.0	85.1
RA(160)	2.00	10.4	51.3

Table 4.1: Performance loss (%) of different programs depending on their memory heaviness, measured in LLC misses per 10^3 instructions

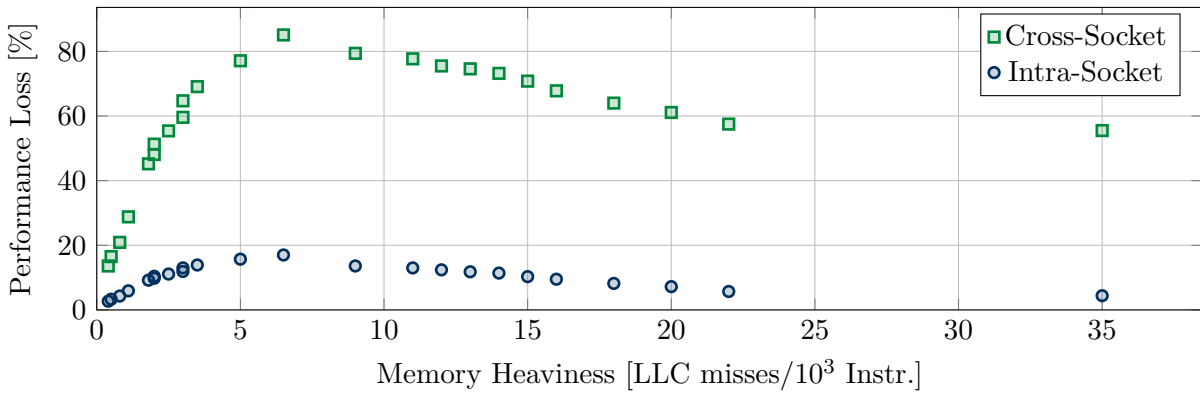


Figure 4.2: Performance loss (%) of the RA program depending on its memory heaviness, measured in LLC misses per 10^3 instructions

an example of highly memory-heavy workloads that are expected to suffer significant performance degradation when executed on non-local memory. Moreover, I evaluate applications with intermediate characteristics using DuckDB queries from the TPC-H benchmark suite [48]. Additionally, I extend the RA program with an adjustable number of computations between memory accesses, allowing the memory heaviness to be artificially reduced stepwise.

For each application, I executed experiments using the HARP framework. First, each application is scheduled on a single NUMA node with its memory bound to the same node using `numactl --membind` [30] to obtain a baseline execution time. Second, the memory is bound to different NUMA nodes on the same socket to measure the performance impact of intra-socket remote memory access. Finally, the memory is bound to NUMA nodes on a different socket to evaluate the impact of inter-socket remote memory access.

4.2.2.4 Observations and results

As shown in Table 4.1, cross-socket memory accesses result in significantly worse performance than intra-socket remote accesses, as expected. However, beyond this general

trend, the results are less consistent than initially anticipated. I made three noteworthy observations.

The first observation is that the EP benchmark occasionally showed a slight performance increase when executed on non-local memory. Since these deviations were small and within the expected measurement noise, I don't analyze them further.

The second observation concerns the RA program with an increasing computational workload and is illustrated in Figure 4.2. The figure depicts the performance loss caused by remote memory accesses in percentage, plotted against the measured memory heaviness. Two measurement series are shown, including one representing memory accesses across SNCs within the same socket, referred to as *Intra-Socket*, and the other representing memory accesses across different sockets, referred to as *Cross-Socket*. While the effects are generally more clearly for cross-socket memory accesses, both series exhibit a similar overall trend.

In the configuration, where the RA program performs no additional computations, the measured memory heaviness is approximately 35 LLC misses/ 10^3 instructions. As the number of computations within the RA program increases, the memory heaviness decreases as intended. However, the corresponding performance loss does not decrease monotonically. Instead, it initially increases gradually, reaches a peak value, and subsequently declines.

The third observation concerns the comparison between DuckDB queries and the RA program configured to achieve the same memory heaviness. Contrary to the expectations, distinct applications with identical memory heaviness exhibited significantly different performance losses when accessing remote memory.

A complete presentation of the results is provided in Table A.1 in the Appendix.

4.2.2.5 Discussion

These observations, particularly the second and third, suggest that NUMA distances and memory heaviness alone are insufficient to accurately predict performance degradation when migrating applications between NUMA nodes. While the initial hypothesis assumed a relatively direct relationship, the experiments indicate that additional factors influence performance.

Due to time constraints, I could not further investigate these factors or determine whether the observed behaviors are architecture-specific or indicative of more general trends. Consequently, while the proposed model provides a promising starting point, further analysis and additional metrics are required to achieve reliable performance predictions.

4.2.2.6 Temporary solution

Assessing the performance impact of scheduling applications across different NUMA nodes constitutes an essential aspect of integrating NUMA awareness within the HARP framework. Without a reliable estimation of the potential performance degradation, it is difficult to comprehensively evaluate both the benefits and the remaining limitations of this integration. As an interim solution, I propose providing expected cross-NUMA

performance losses through the application-specific mapping file, effectively relying on the client-side approach described in Section 4.1.3. This method is not suitable as a long-term solution, since it requires prior measurements for each application managed by HARP. However, for the purposes of this work, I collected performance data for a representative set of applications to enable at least an initial evaluation of the effectiveness of the NUMA awareness integrated into HARP.

4.2.3 MigratePages feature

As proposed in Section 4.1.4.2, I realize the migration of memory on the client side and implement it using HARP's **MappingFeature**. To perform memory migration for a given application, I employ the Linux `migrate_pages()` [28] system call. This system call transfers all memory pages of a given process residing on a set of source NUMA nodes (`from_nodes`) to a set of destination NUMA nodes (`to_nodes`).

To apply this mechanism, it is necessary to determine the NUMA nodes on which the application was previously executing and the nodes to which its threads are now assigned. Each time a `mapping_update()` is invoked, the client retrieves information about the newly allocated cores. Using helper functions exposed through the updated mapping, I identify the NUMA nodes corresponding to these cores. Finally, with knowledge of both the previous and current NUMA node allocations, I invoke the `migrate_pages()` system call to move the application's memory accordingly, completing the migration process.

5 Evaluation

This chapter presents a comprehensive evaluation of the contributions introduced in the previous chapters, with the goal of assessing their effectiveness and impact. Chapter 3 described the integration of DuckDB into HARP, aiming to enable efficient query execution, while Chapter 4 focused on extending HARP with NUMA awareness to improve HARP’s scheduling decisions.

In the following, I start with analyzing whether the integration of DuckDB into HARP can improve query execution performance and energy consumption. Subsequently, I examine whether the NUMA-aware extension of HARP successfully performs NUMA-conscious application placement and analyze how this placement strategy affects overall application performance.

5.1 Integration of DuckDB into HARP

In this section, I evaluate the impact of integrating DuckDB into the HARP framework, with a particular focus on its effects on performance and energy efficiency. To provide the necessary context, I first describe the measurement setup, outlining the system configuration used during the experiments to establish an understanding of the test environment. Following this, I introduce the benchmark methodology employed to assess the DuckDB integration. Finally, I analyze the experimental results to determine how the integration into HARP influences DuckDB’s query execution time and energy consumption.

5.1.1 Measurement setup

For the evaluation of the DuckDB integration into HARP, I use a heterogeneous system, as HARP is specifically designed to improve scheduling performance on systems with heterogeneous CPU architectures.

I conduct the experiments on an Intel Raptor Lake Core i9-13900K processor, which provides eight P-cores supporting Simultaneous Multi-Threading (SMT) [1] and 16 E-cores without SMT, resulting in a total of 32 logical cores. The system is equipped with 128 GiB of main memory and runs a Linux 6.4 kernel.

5.1.2 Benchmarks

The set of DuckDB queries used in the experiments is derived from a variation of benchmarks provided by the DuckDB benchmark suite [13]. These include queries based on the TPC-H decision support benchmark, executed with a scaling factor of 10 [46]. TPC-H consists of a collection of business-oriented ad hoc queries and concurrent data

modifications designed to represent decision support systems operating on large data volumes.

In addition to TPC-H, the DuckDB-related benchmarks include *Cast* queries, which primarily focus on data type conversions within query execution, and queries from the ClickBench benchmark [8]. ClickBench is designed to represent typical workloads encountered in areas such as clickstream and traffic analysis, reflecting common ad hoc analytics and real-time dashboard use cases.

To assess scheduling interactions and system-level performance effects, all DuckDB queries are evaluated both individually and in combination, either with each other or alongside additional applications. Among them is AlexNet [25], a convolutional neural network for image recognition, which serves as a representative workload for deep learning inference.

Furthermore, two benchmarks from the NPB suite [3], version 3.4.2, are included to evaluate parallel scheduling behavior under different communication characteristics. The first, EP, represents an Embarrassingly Parallel workload that is compute-intensive and requires virtually no interprocessor communication. The second, MG, is a simplified MultiGrid calculation characterized by highly structured long-distance communication patterns, making it suitable for evaluating scheduling influences under communication-intensive scenarios. For both benchmarks, I use the problem size class C.

This diverse collection of benchmarks ensures that the evaluation covers a broad spectrum of workloads, ranging from deep learning inference to parallel high-performance computing, thereby enabling a comprehensive assessment of DuckDB queries.

5.1.3 Experimental results

The evaluation of the integration of DuckDB into the HARP framework focuses on measuring both performance and energy consumption using the presented Intel Raptor Lake system as the test platform. The experiments compare HARP against the default Linux scheduler CFS, which serves as the baseline for all measurements.

To comprehensively assess the effects of the integration, the experimental scenarios are designed to cover two distinct execution contexts. The first context considers the execution of a single DuckDB application, where only one DuckDB program runs on the system. The second context considers a mixed workload environment, where a DuckDB query is executed concurrently with additional workloads. These include either further DuckDB queries or other applications such as AlexNet and selected benchmarks from the NPB, as introduced in the previous section, in order to analyze HARP’s scheduling behavior in multi-application scenarios.

For each of these contexts, I measure both the overall execution time and the total energy consumption. Each experiment runs 10 times, and the reported results represent the average values, accompanied by error bars to indicate variability across runs. To facilitate a meaningful comparison, the results are expressed as the percentage improvement of HARP relative to CFS. A reported value of 0 % indicates that HARP performs on par with CFS, while a value of 10 % reflects a 10 % reduction in execution time or energy consumption compared to CFS. Conversely, a negative value, such as −10 %, indicates a 10 % performance degradation or increased energy consumption relative to the baseline.

For the experiments, I evaluate HARP in two distinct modes. In the offline mode, the scheduler makes all scheduling decisions solely based on pre-generated OPs without performing further measurements, whereas in the online mode, HARP actively measures performance and energy at runtime to generate the OPs dynamically. Since these approaches may lead to different performance and energy efficiency characteristics during runtime, I evaluate both modes separately in the experiments to capture their respective impacts on DuckDB query execution.

5.1.3.1 Offline HARP

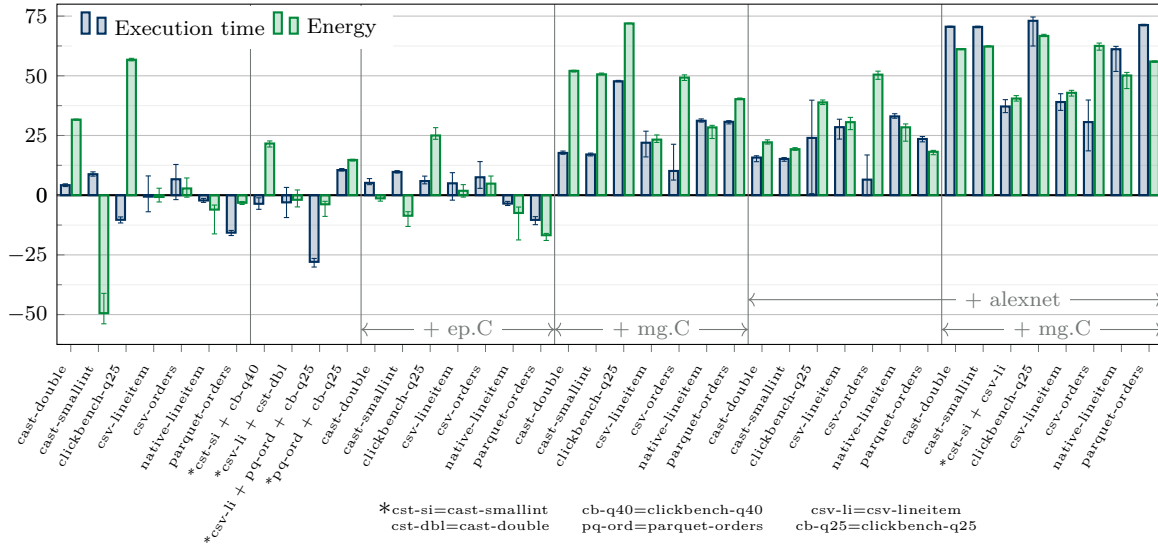


Figure 5.1: Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and rows per second as performance metric.

When using HARP with pre-generated OPs, they capture the performance and energy characteristics for all possible combinations of P-cores and E-cores to guide its scheduling decisions. The corresponding results are presented in Figure 5.1.

Executing single DuckDB queries The results of executing the single DuckDB applications are shown on the left-hand side of Figure 5.1. Overall, the outcomes vary significantly, with the average energy consumption ranging from an improvement of 57 % to a degradation of 49 %, while execution time ranges from a 9 % improvement to a 16 % degradation.

A particularly notable case is q25 from the ClickBench benchmark, which shows a 57 % improvement in energy efficiency. During the preceding OP generation, HARP identified that the performance of this query does not scale with the number of available cores. Consequently, it favored a configuration utilizing only two P-cores, which results in a 10 % slower execution time compared to CFS. However, HARP aims to balance performance and energy efficiency. In this case, the significant reduction in energy

consumption outweighs the performance degradation, resulting in an overall positive outcome. I evaluated two additional ClickBench queries which exhibit a similar behavior, as illustrated in Figure A.1 in the Appendix. More generally, for all diagrams presented in the evaluation of the DuckDB integration into HARP, a corresponding diagram is provided in the Appendix that includes additional scenarios omitted from the main text due to space constraints.

In contrast, the `smallint` query from the Cast benchmark performs notably worse, showing an energy consumption increase of 49 %, which cannot be compensated by its modest 9 % performance improvement. Interestingly, the `double` query from the Cast benchmark demonstrates improvements in both execution time and energy efficiency. Although these two queries differ only in the data type being cast, HARP’s scheduling decisions lead to completely different resource allocations and, consequently, very different energy characteristics. For the `double` query, HARP allocated 14 cores, whereas for the `smallint` query, the scheduler used the full 32 cores. A plausible explanation is that the `smallint` query does not benefit sufficiently from the higher degree of parallelism to offset the additional energy required for utilizing more cores.

The remaining single DuckDB benchmarks originate from the TPC-H suite. Among these, only the `csv_orders` query exhibits minor improvements in both performance and energy efficiency. For all other TPC-H queries, HARP performs either similar to or worse than CFS for both, execution times and energy consumption.

For the geometric mean across all single-application scenarios, HARP shows an overall 1.7 % degradation in execution time and 0.4 % degradation in energy consumption. Although these values are near 0 % and therefore close to the results observed with CFS, the outcomes are generally worse than expected.

This becomes particularly evident when comparing these results to prior evaluations of HARP in other application domains [45]. For example, when running single benchmarks from the NPB or image recognition models, previous experiments reported average improvements of approximately $1.22\times$ in performance and $1.44\times$ in energy efficiency. Consequently, the advantages of HARP for DuckDB workloads appear considerably weaker than for these previously tested application types.

Multi-application scenarios The results of executing a DuckDB query alongside additional applications are presented in the remaining parts of Figure 5.1. The evaluated scenarios are grouped based on the type of application that is executed concurrently with the DuckDB query.

The first group comprises scenarios where a DuckDB query is executed alongside other DuckDB queries. For the combination of the `smallint` query from the Cast benchmark and `q40` from the Clickbench benchmark, the overall outcome appears to be dominated by the behavior of `q40`. The significant benefits in energy consumption and the moderate performance degradation observed when running `q40` individually also carry over to this combined scenario, outweighing the degraded energy consumption and the performance improvement of the `smallint` query.

Although `q25` is assumed to behave similar to `q40`, a different trend can be seen when combining the `parquet_orders` query from the TPC-H suite with `q25`. While both

queries individually show performance degradation compared to CFS, their combination achieves better results for both performance and energy consumption. This demonstrates that HARP is able to effectively coordinate resource allocation between these two queries.

In contrast, the other two combinations within this group result in degradations in both performance and energy consumption. I observed that HARP appears to face challenges in determining core assignments that avoid core sharing between these DuckDB queries, which resulted in suboptimal core allocations and degraded overall efficiency.

In general, it is important to recall from Section 3.2.1.3 that adapting the runtime parallelism of a DuckDB program is only feasible at the granularity of entire queries. Consequently, when HARP modifies the resource allocation of a DuckDB application during the execution of a query, the internal thread and task management of DuckDB can only be adjusted once that query has completed. This limitation becomes particularly relevant in the multi-application scenarios. For example, when a DuckDB application registers first at the HARP RM, it receives an initial resource allocation. If shortly afterward another application registers, the RM may adapt the allocation of the DuckDB application. If the DuckDB query is already running, the internal parallelism cannot be adapted immediately, meaning that the query may continue to execute with suboptimal parallelization, such as multiple threads being bound to a single core. Such a mismatch in resource allocation has the potential to significantly degrade performance. The negative impact is especially pronounced for long-running queries, such as those in the TPC-H benchmark suite used in the experiments. Overall, this constitutes a general performance degradation risk for DuckDB queries in multi-application scenarios managed by HARP.

The second group of scenarios evaluates the execution of a DuckDB query alongside the EP benchmark from the NPB. Here, a notable pattern emerges, where, except for the combination involving q40 from Clickbench, HARP appears to have difficulty with finding a suitable core allocation. Since EP is an Embarrassingly Parallel benchmark that scales nearly linearly with the number of available cores, it benefits most when it gets assigned all cores. This inherently conflicts with DuckDB queries that also require a number of cores, leading to scheduling trade-offs. In the case of the q40 query, this issue does not arise, as this query only requires two P-cores, making it easier for HARP to find a suitable allocation.

In general, when combining DuckDB queries with EP, performance tends to improve modestly compared to CFS, whereas energy consumption mostly worsens. This trade-off appears intuitive, as combining DuckDB queries with a highly compute-intensive workload like EP inherently increases the potential for performance improvements under effective scheduling, while simultaneously posing a higher risk of elevated energy consumption.

The remaining groups include scenarios where DuckDB queries are executed alongside the MG benchmark from NPB, AlexNet, or both simultaneously. In these cases, HARP consistently outperforms CFS in terms of both energy consumption, with improvements of up to 67%, and execution time, reaching gains of up to 73%. These results indicate that MG and AlexNet benefit substantially from HARP’s scheduling strategy, often dominating the performance and energy characteristics of these mixed workloads. Particularly in scenarios where both MG and AlexNet are executed alongside a DuckDB query, the improvements are most pronounced, suggesting that HARP effectively exploits the underlying workload characteristics to optimize scheduling decisions in these cases.

Considering the geometric mean across all multi-application scenarios, HARP achieves a reduction in execution time of 21 % and a decrease in energy consumption of 28 %. These improvements are considerably larger than those observed in the single-application evaluations, which is unsurprising given that the performance gains in scenarios involving MG and AlexNet strongly dominate the multi-application results.

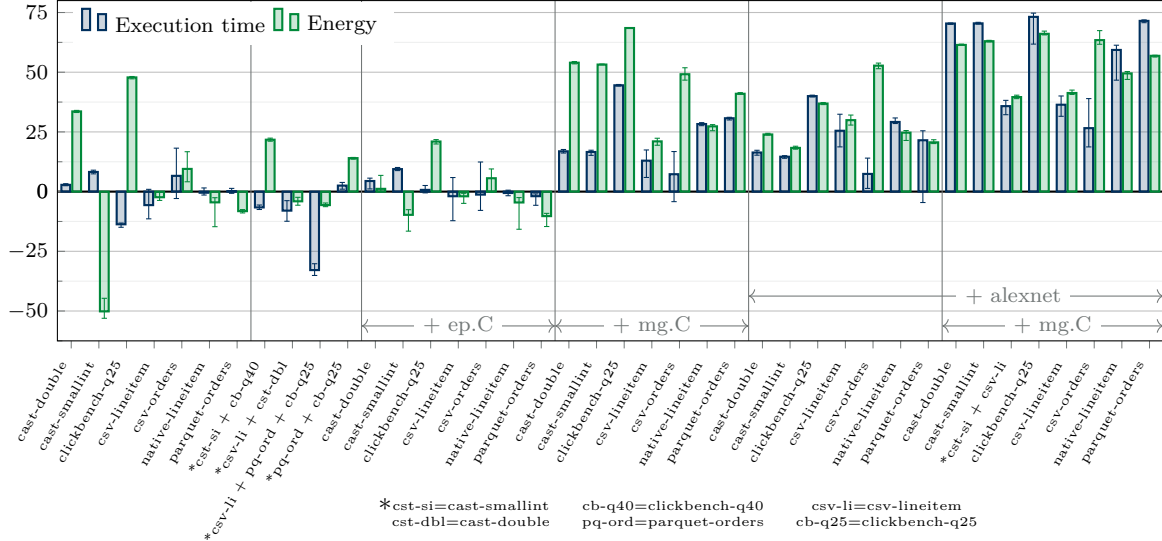


Figure 5.2: Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and IPS as performance metric.

IPS as performance metric After analyzing the behavior of DuckDB queries when executed separately or alongside other applications, the question remains why single DuckDB applications benefit less from HARP’s scheduling approach compared to the performance improvements observed in previous measurements for other application types [45]. One possible explanation could lie in the choice of the performance metric used to guide HARP’s scheduling decisions.

In the previous experiments, Smejkal et al. employed IPS as performance metric and used it in HARP’s optimization process. In contrast, the integration of DuckDB into HARP enables the use of database-specific performance indicators. As described in Section 3.2.2.3, I decided to utilize rows processed per second as performance metric, based on the hypothesis that an application-specific metric could provide better insights into database query behavior than a generic hardware-level metric such as IPS.

However, it is conceivable that the number of processed rows per second is less meaningful for effective scheduling than initially expected. To investigate this hypothesis, I repeated the complete set of measurements using IPS as the guiding performance metric instead of rows per second. The corresponding results are presented in Figure 5.2.

Overall, I could not observe substantial improvements compared to the results in Figure 5.1. In fact, IPS appears to yield slightly worse scheduling decisions in some cases, such as for q25 from the ClickBench benchmark.

Consequently, the limited benefits of integrating DuckDB into HARP cannot be attributed to the choice of the performance metric. At this point, I can't provide a conclusive explanation for the modestly degraded performance and energy consumption. A plausible assumption is that DuckDB workloads, due to their execution characteristics and resource usage patterns, may inherently benefit less from HARP's optimization approach than the application types examined in the previous study. Further investigations would be required to confirm this hypothesis.

5.1.3.2 Online HARP

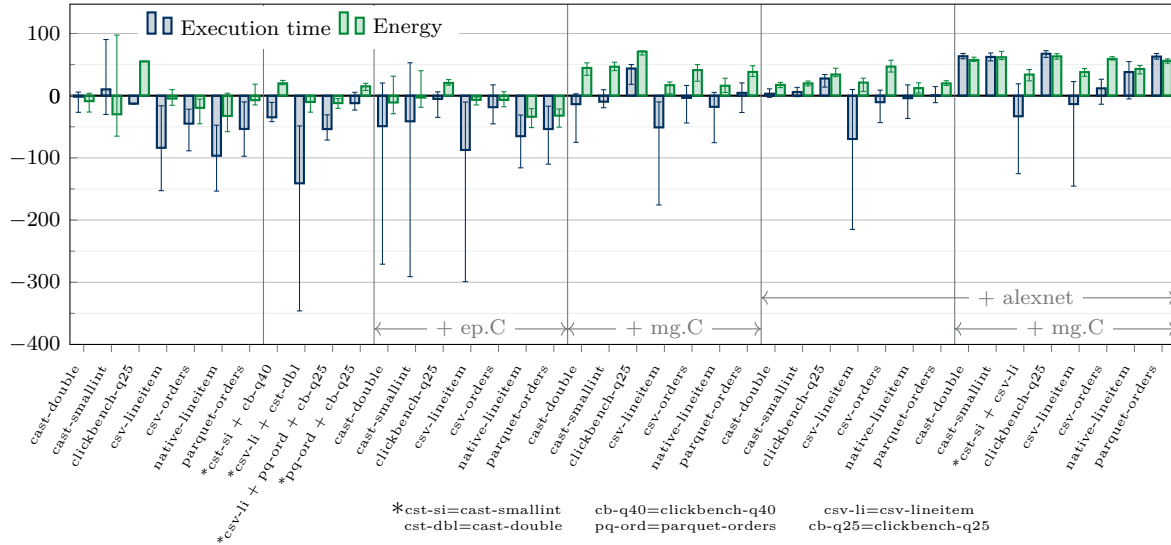


Figure 5.3: Relative improvement in % of HARP over CFS. HARP runs with online-generated operating points and rows per second as performance metric.

In contrast to the experiments presented in the previous section, the following results are obtained using the online mode of HARP. In this mode, HARP begins with a warm-up phase during which it frequently tests different configurations until a predefined number of OPs have been generated. These OPs serve as a starting point for guiding scheduling decisions during the subsequent measurements.

When a scenario involves multiple applications, each application undergoes its warm-up phase independently. This ensures that HARP has a representative set of OPs for each application before the actual combined evaluation begins. Once the warm-up phase is completed for all applications in the scenario, the measurement phase starts. During this phase, HARP uses the previously generated OPs as an initial basis for scheduling, while continuing to perform additional online measurements to refine its decisions dynamically as the workload progresses.

The results of these experiments are presented in Figure 5.3.

Executing single DuckDB queries I begin by examining the execution of individual DuckDB queries in the online mode. The first observation focusses on the large error bar ranges, reaching up to 137 % for performance and 162 % for energy consumption. This result is not unexpected, given the nature of the online mode, where performance and energy are continuously measured, while HARP dynamically changes resource allocations over time.

In comparison to the offline mode, the overall values are generally worse. This can be attributed to the fact that DuckDB’s internal parallelism can only be adapted at query granularity. As a result, the warm-up phase is insufficient for HARP to acquire a comprehensive understanding of DuckDB’s execution behavior, as discussed in Section 3.2.1.1.

This effect is particularly pronounced for long-running queries, such as those from the TPC-H benchmark suite, which in general exhibit notably worse results. In the geometric mean, performance degrades by 61 % compared to CFS, while energy consumption worsens by 10 %.

In general, it is not surprising that online mode tends to yield worse performance results than CFS, as previous experiments with other applications have also demonstrated performance degradations in the online mode [45]. Nevertheless, the expectation is typically that these losses are compensated by energy improvements. In the case of DuckDB, however, the degradation in energy consumption is less severe than the performance degradation, but still persists. Taken together, the results reveal a substantial performance drawback, particularly emphasizing DuckDB’s limited adaptability when executing long-running queries and highlighting the importance of enabling more frequent scaling updates for DuckDB in future research.

Executing multiple applications When examining the multi-application scenarios, the variability in performance becomes even more pronounced, with error bars reaching ranges of up to 344 %. Although the energy consumption exhibits smaller fluctuations, the ranges of up to 60 % still exceed those observed in the offline mode.

The general trend observed in Section 5.1.3.1 in the offline measurements — where the MG and AlexNet applications dominate performance and energy consumption — remains apparent in the online scenarios. Furthermore, combinations of DuckDB queries with EP contribute to the observed large error bars. As discussed in the context of the offline measurements, HARP encounters difficulties when scheduling the EP benchmark, which requires substantial resources, alongside competing DuckDB workloads. The online mode exacerbates this effect, as HARP explores multiple resource allocation combinations, resulting in both highly efficient and highly suboptimal allocations.

Overall, the geometric mean across the multi-application scenarios indicates a performance degradation of 19 % and an improvement in energy consumption of 21 %, largely due to the dominating influence of AlexNet and MG. When considering both metrics jointly, this yields a modest overall improvement for the multi-application scenarios. Nevertheless, the limitations of DuckDB’s internal adaptation mechanisms continue to produce less favorable outcomes than might be expected based on prior measurements [45].

5.2 NUMA Awareness Extension

In this section, I examine the impact of extending HARP with NUMA-aware scheduling capabilities. The primary focus lies on evaluating whether the NUMA-aware extension correctly influences application placement across NUMA nodes. In addition, I analyze how this placement strategy affects application performance. Since the Intel Raptor Lake system used in the previous experiments does not feature a NUMA architecture, these evaluations are conducted on a different system, which I introduce separately in the next section. Subsequently, I describe the selection of test applications employed for the experiments. Finally, I analyze the experimental results to assess the effectiveness of the NUMA-aware extension in improving application placement and examine its influence on performance outcomes.

5.2.1 Measurement setup

The experiments in this section are conducted on a dual-socket server equipped with two Intel Xeon Max 9468 processors. Each socket is partitioned into four SNCs, and each SNC comprises 64 GiB of main memory and 12 P-cores with SMT support, resulting in a total of 192 logical cores across the system. The NUMA distances of the system reveal that memory accesses across SNCs within the same socket incur approximately 20 % higher latency, while accesses across different sockets exhibit an increased latency of about 110 %. For completeness, it should be noted that the system also includes eight additional NUMA nodes with high-bandwidth memory but without further cores. However, these nodes are not utilized in the experiments and are hence not considered in the experimental evaluation. All experiments are performed on a Debian Testing distribution running a Linux 6.6 kernel.

5.2.2 Benchmarks

To evaluate the effectiveness of the NUMA-aware scheduling extension, I use the RA program introduced in Section 4.2.2.3. Specifically, I consider two variants of the RA program. The first one is RA(40), which performs 40 computations and has been shown to be highly sensitive to NUMA placement, as shown in Table 4.1 in Chapter 4. The second one is RA(1000), which executes 1000 computations and exhibits lower sensitivity to NUMA placement, as shown in Table A.1 in the Appendix. These two variants allow for an assessment of how the NUMA-aware extension influences both placement and performance under workloads with differing NUMA sensitivity.

5.2.3 Experimental results

The evaluation of the NUMA awareness extension is conducted on the dual-socket system presented in Section 5.2.1, with the analysis first focusing on application placement decisions and subsequently on the resulting performance impact. Since the system operates with Linux kernel version 6.6, the measurements are compared against the EEVDF scheduler rather than CFS, as EEVDF has become the default scheduler in this

kernel version. However, both schedulers are generally comparable in terms of NUMA awareness, so this substitution is not expected to affect the results [9].

For the evaluation, the two RA programs specified in the previous section are considered across four distinct scenarios. The first, referred to as the *2-Apps* scenario, executes two instances of the RA(40) program, each requesting 64 threads, and is used to evaluate whether HARP can effectively avoid cross-socket memory accesses. The second, the *8-Apps* scenario, runs eight instances of RA(40), each requesting 14 threads, verifying if NUMA-local memory locality is preferred over socket-local and cross-socket memory accesses. The third, the *3-Apps* scenario, combines two RA(40) instances with a single RA(1000) instance, all requesting 64 threads, and investigates not only memory locality but also how HARP handles competing topology demands, determining which applications receives better locality. Finally, the *12-Apps* scenario includes six RA(40) and six RA(1000) instances, each requesting 14 threads, allowing the evaluation of HARP’s ability to distribute memory locality and manage competing topology requirements at a larger scale.

All scenarios are evaluated using the offline mode of HARP. To isolate the impact of NUMA-aware placement, each scenario is provided with only a single OP representing the configuration of either 64 or 14 threads, rather than with OPs for all possible combinations of P-cores. This configuration constrains HARP’s assignments to a fixed number of threads, allowing the study to focus exclusively on the effects of NUMA-aware placement.

5.2.3.1 Placement

In the following section, I evaluate how EEVDF and HARP assign cores to applications in a NUMA system. All scenarios are executed 10 times to obtain meaningful average execution times. However, the placement diagrams presented correspond to a single representative run.

For all scenarios, I visualize core assignments using Gantt diagrams, which illustrate the temporal allocation of applications across the system’s NUMA nodes. In these diagrams, the x -axis represents elapsed time in seconds, while the y -axis enumerates the individual NUMA nodes and their constituent cores. Each horizontal bar corresponds to a specific core, and the color of the bar encodes the identity of the application thread assigned to that core. The length of each colored segment indicates the duration for which the corresponding thread ran on the assigned core.

For EEVDF, I employ the `ftrace` framework [39], as part of the Linux kernel’s tracing infrastructure, to record the points in time when processes and threads are created and terminate, as well as all events in which a thread migrates from one core to another. For HARP, I obtain placement information through debug outputs of the HARP RM that report thread-to-core assignment decisions and the start and finish times of applications.

2-Apps scenario In the following, I first analyze the 2-Apps scenario. The scheduling behavior of the default EEVDF scheduler is illustrated in Figure 5.4. From the beginning of execution until approximately 7s, only a single thread from each RA(40) instance is active. This is the thread, which is responsible for allocating the memory used in

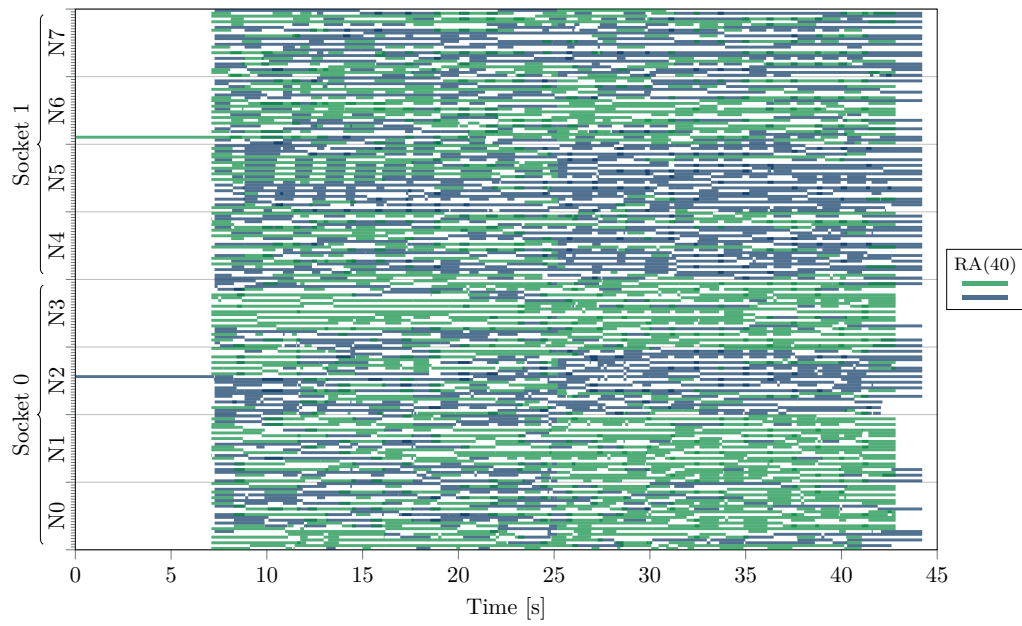


Figure 5.4: Thread-to-core assignments for the 2-Apps scenario managed by EEVDF

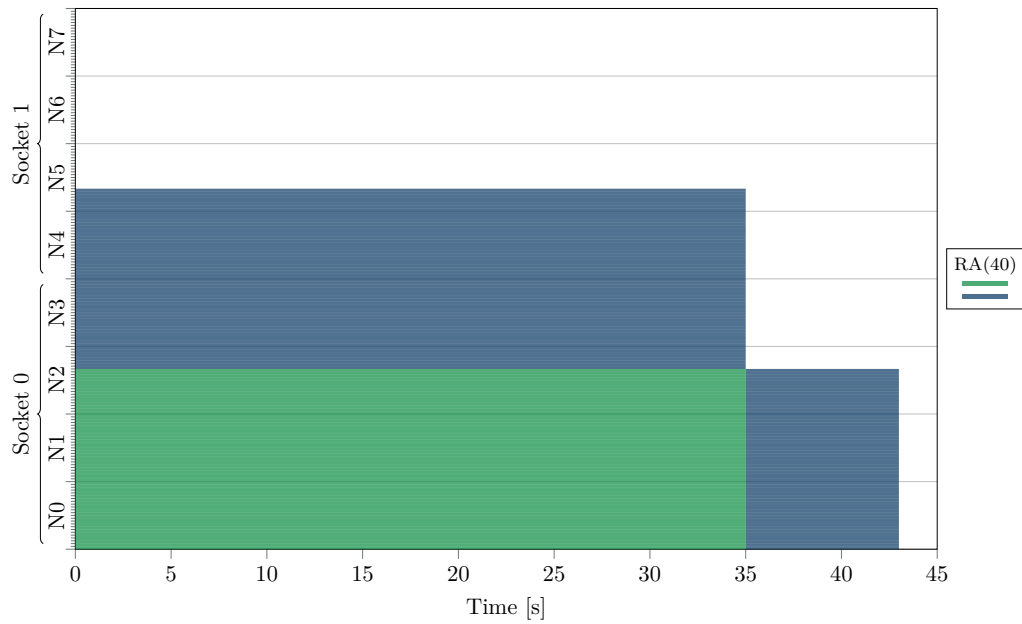


Figure 5.5: Thread-to-core assignments for the 2-Apps scenario managed by HARP without NUMA awareness

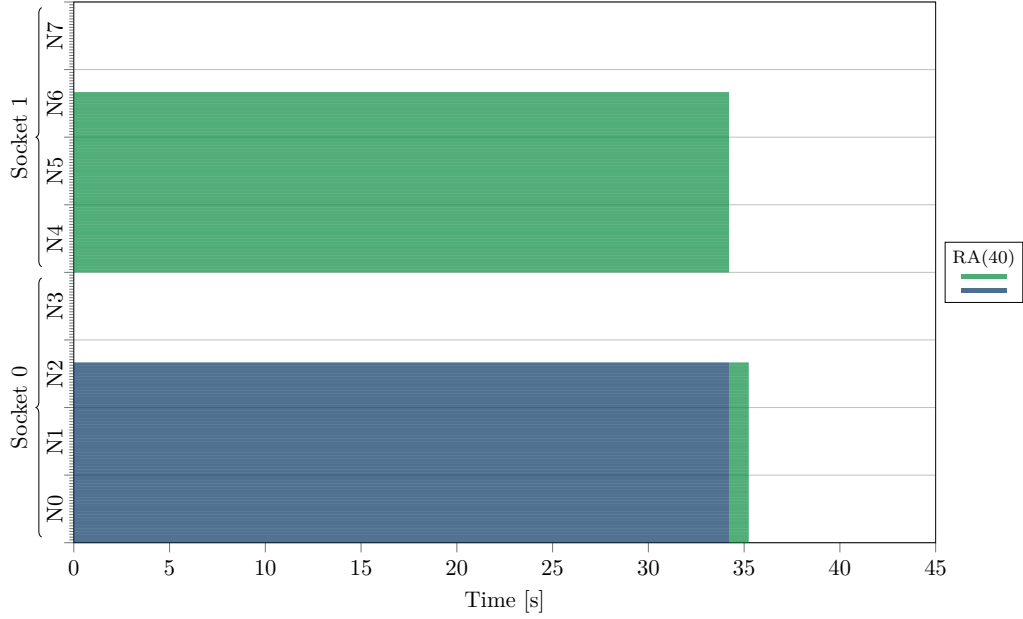


Figure 5.6: Thread-to-core assignments for the 2-Apps scenario managed by HARP with NUMA awareness

the random access operations. Until 25s, EEVDF shows no clear strategy for placing threads on specific NUMA nodes, as threads from both program instances are widely interleaved across all nodes. However, in the subsequent part of the diagram, a more structured behavior emerges. Most threads of the first instance are scheduled on NUMA nodes 0, 1, 3, and 6, while the majority of threads of the second instance are placed on NUMA nodes 2, 4, and 7. This indicates that EEVDF appears to recognize NUMA effects to some extent and incorporates this knowledge into its scheduling decisions. Nevertheless, a closer inspection of the diagram reveals that, at certain points, threads from both program instances are even scheduled on the same logical cores. While technically possible, this behavior is generally not favored, as it can introduce contention and degrade performance [22].

The scheduling behavior of HARP without NUMA awareness is shown in Figure 5.5. In this configuration, HARP follows its default placement strategy. Unlike CFS, which initially allocates only the two threads required for memory allocation, HARP immediately allocates the full number of threads requested by each application, regardless of whether all threads are actively used at the start. Consequently, the first scheduled application directly receives the first 64 available cores, and the second application is assigned the next 64 free cores, resulting in two clearly separated, contiguous thread-to-core assignment blocks. However, because the second application is placed across NUMA nodes 3, 4, and 5, its threads are distributed across both sockets. Given that memory for the RA program is allocated at program startup using a single thread, and no special NUMA-aware data placement strategy is employed, it can be assumed that all memory is allocated on the NUMA node associated with the initializing thread. Consequently, a

significant portion of the threads from the second application probably performs memory accesses across sockets, which are typically considerably slower than intra-socket memory accesses.

Once the first application terminates, HARP triggers a rescheduling phase, moving the second application to the first 64 cores, which are located on a single socket. This potentially reduces cross-socket memory accesses and speeds up the remaining execution, albeit at the cost of additional overhead caused by the thread migration itself.

Figure 5.6 shows the scheduling behavior of HARP with NUMA awareness enabled. The first application again receives the first 64 cores on the first socket. However, in contrast to the NUMA-unaware configuration, HARP now recognizes that the second application, requesting 64 threads, can be placed entirely on the second socket, which has sufficient free cores available. HARP therefore assigns the second application to the first 64 cores of the second socket, ensuring that all threads of the application remain within the same socket and thus avoiding cross-socket communication during execution. While this represents a significant improvement in placement decisions, an aspect remains that is still suboptimal. When the first application finishes, HARP reschedules the second application, migrating it to the now-available first 64 cores on the first socket. This behavior is suboptimal, as the data allocated at program initialization likely resides in the memory of the second socket, resulting in memory accesses across sockets. Additionally, the migration introduces context-switching overhead, which occurs even in situations where no context switch would be beneficial. This is in general a problem currently not accounted for in either the baseline HARP scheduler or the NUMA-aware extension.

During this work, I implemented the `MigratePages` feature, as described in Section 4.2.3, which enables the migration of application-specific data to the newly assigned NUMA nodes after rescheduling. Since the functionality is implemented on the client side, it is only available to applications that explicitly make use of it. As the RA program employed in these experiments does not currently support this feature, it is not applicable in the present evaluation. Enabling broader use of this feature could be highly beneficial in future work, particularly for workloads that are sensitive to data locality and memory placement.

8-Apps scenario The scheduling behavior of EEVDF for the 8-Apps scenario is illustrated in Figure 5.7. The visualization reveals a highly interleaved and irregular distribution of threads across cores, making it difficult to derive any meaningful placement strategy from EEVDF’s behavior. Consequently, I focus on comparing HARP with and without NUMA awareness.

Figure 5.8 shows the scheduling behavior of HARP without NUMA awareness. Similar to the previous 2-Apps scenario, HARP applies its default placement strategy. For the first application, it assigns the first 14 free cores, for the second application the next 14, and so forth. This naive allocation leads to several inefficiencies. While only one instance has its threads distributed across sockets, three additional applications still experience thread placements across NUMA nodes within the same socket. Since cross-node memory accesses are generally slower than local accesses, this placement strategy is suboptimal, even though it is less detrimental than cross-socket communication. Importantly, these

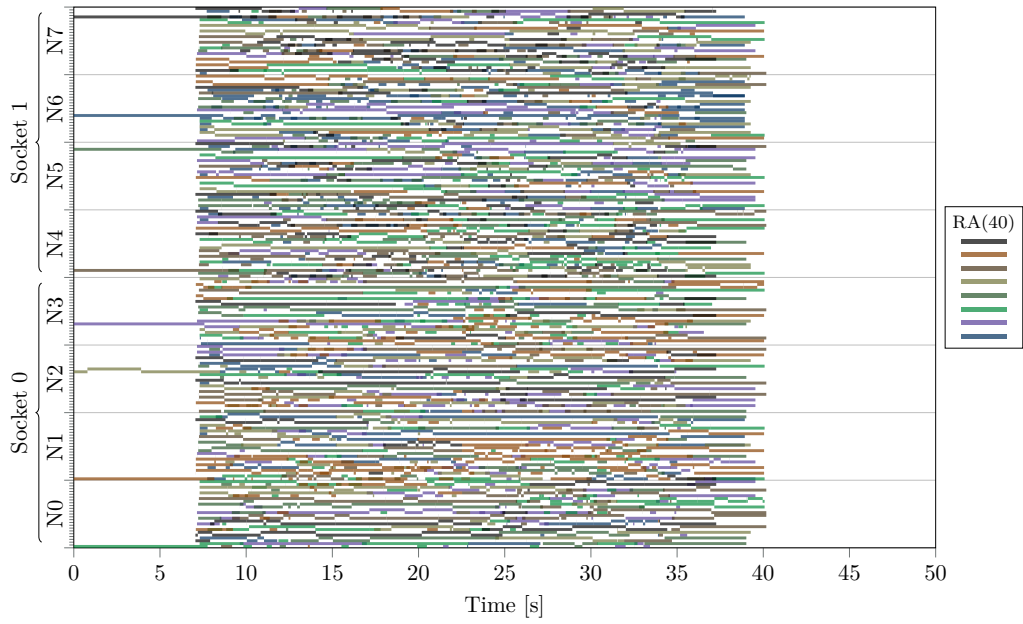


Figure 5.7: Thread-to-core assignments for the 8-Apps scenario managed by EEVDF

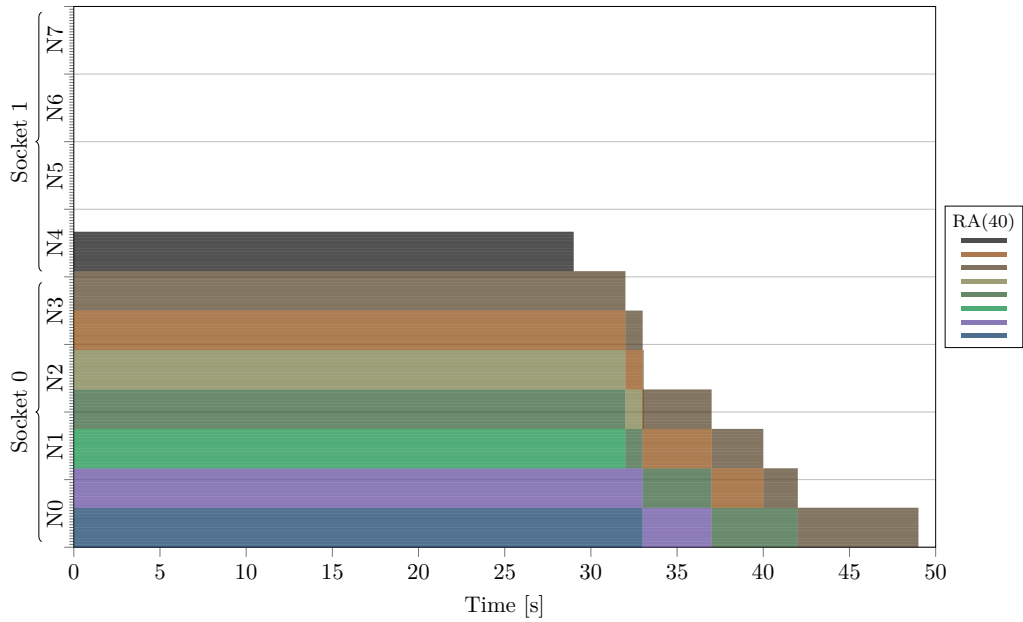


Figure 5.8: Thread-to-core assignments for the 8-Apps scenario managed by HARP without NUMA awareness

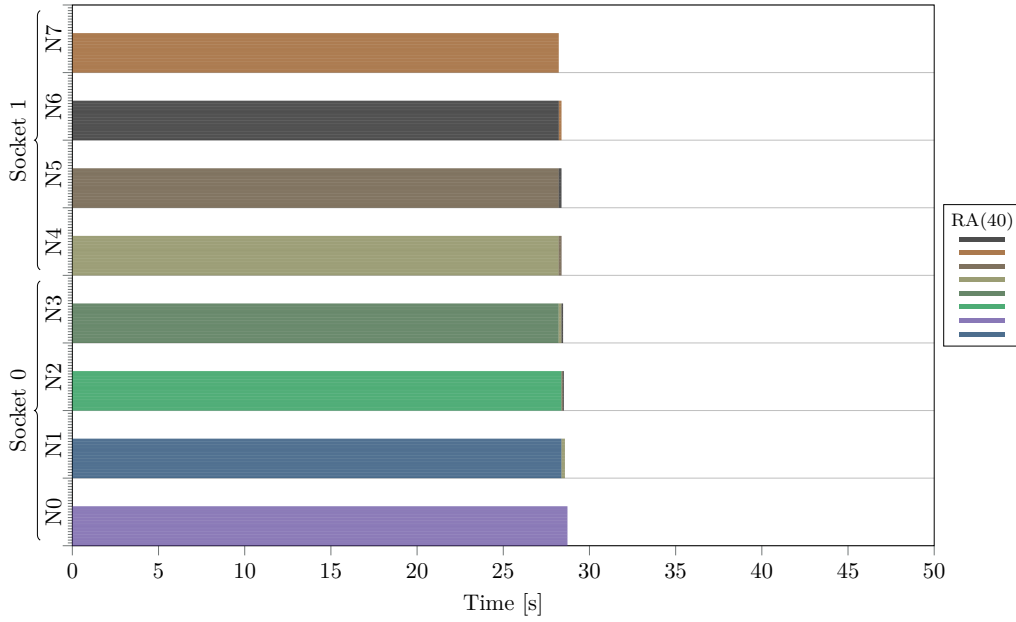


Figure 5.9: Thread-to-core assignments for the 8-Apps scenario managed by HARP with NUMA awareness

inefficiencies could be avoided in this scenario, as the system comprises eight NUMA nodes, each providing 24 logical cores — sufficient to place each of the eight instances entirely within a single node.

Another observation from Figure 5.8 is that the step-like rescheduling behavior becomes even more pronounced compared to the 2-Apps scenario. When one application finishes, HARP immediately reallocates freed cores to other running applications, leading to significant shifts in thread placement during execution.

Finally, Figure 5.9 illustrates the scheduling behavior of HARP with NUMA awareness enabled. Here, HARP correctly identifies that each of the eight RA(40) instances can be fully allocated to a dedicated NUMA node. As a result, the applications are distributed evenly across the eight nodes, avoiding both cross-node and cross-socket memory accesses.

3-Apps scenario The 3-Apps scenario represents the first scenario in which each individual instance could, in principle, be scheduled entirely on a single socket. However, since the system provides only two sockets with 96 logical cores each, it is not feasible to schedule all three instances socket-local. Consequently, at least one of the applications must inevitably be distributed across sockets.

The scheduling behavior of EEVDF is depicted in Figure 5.10. During the initial 56 s, the scheduler distributes threads from each application across all available nodes, resulting in no discernible placement pattern. After this period, the RA(40) instances complete, leaving only the RA(1000) threads active. It is generally observed that RA(1000) executes more slowly than RA(40) under identical conditions, a trend that is

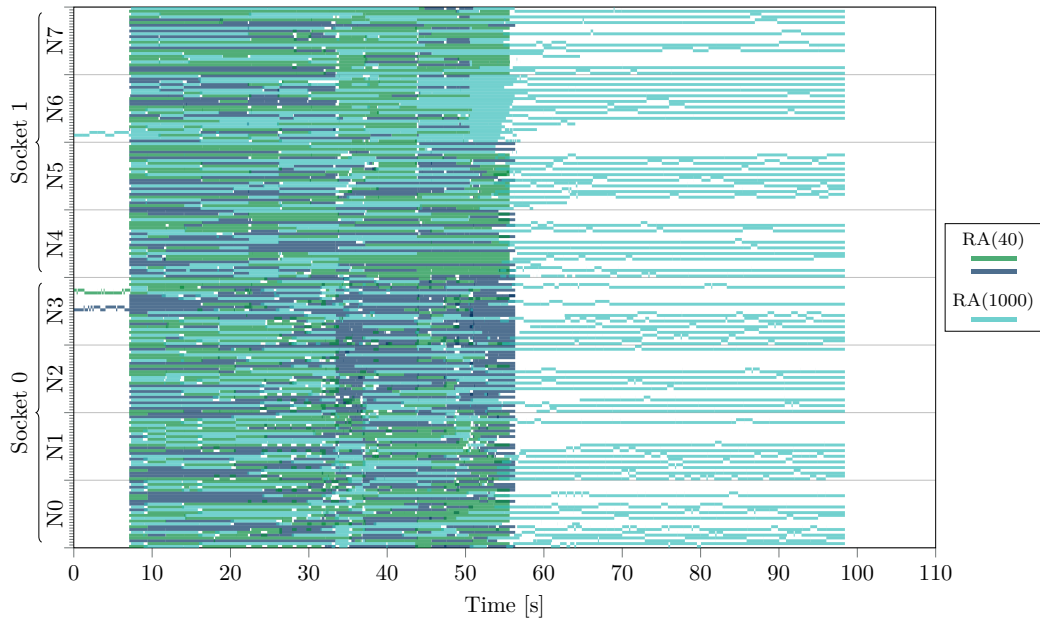


Figure 5.10: Thread-to-core assignments for the 3-Apps scenario managed by EEVDF

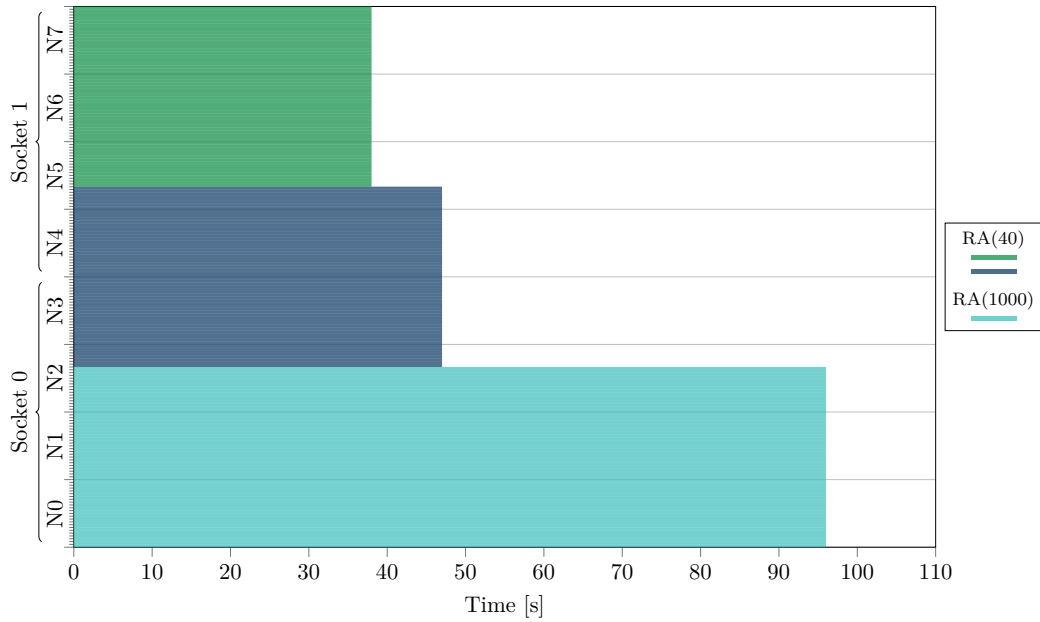


Figure 5.11: Thread-to-core assignments for the 3-Apps scenario managed by HARP without NUMA awareness

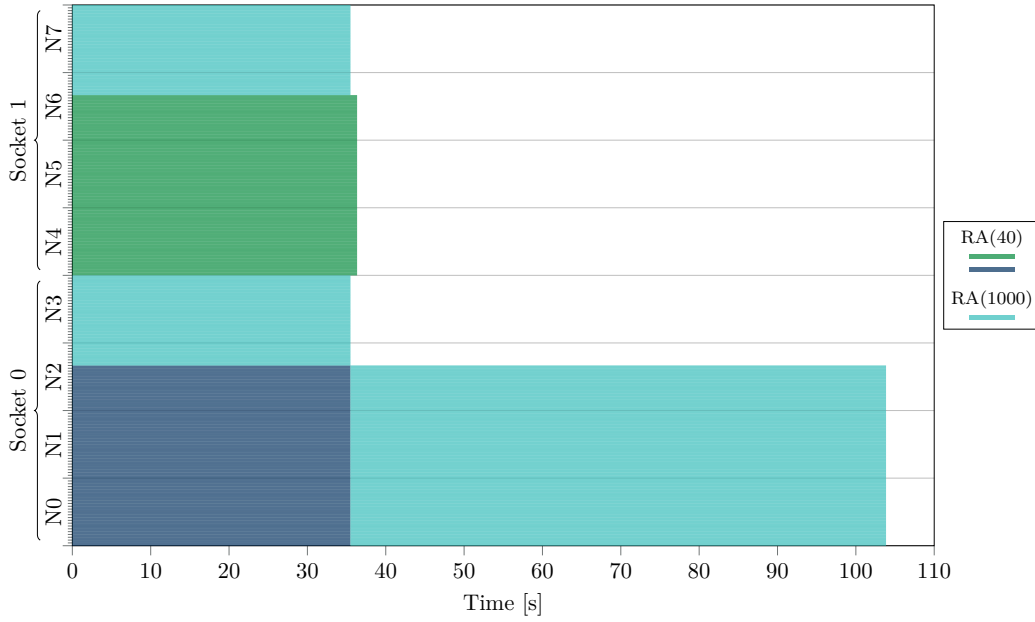


Figure 5.12: Thread-to-core assignments for the 3-Apps scenario managed by HARP with NUMA awareness

also reflected in the subsequent figures. Despite the termination of RA(40), the remaining RA(1000) threads continue to exhibit no clear placement pattern.

The scheduling behavior of HARP without NUMA awareness is shown in Figure 5.11. As in the previous scenarios, HARP demonstrates its naive allocation strategy. In this case, this results in two applications being scheduled socket-local, while the third is distributed across both sockets. However, because HARP assigns each set of 64 cores in the order of application registration, it is effectively random which application receives the first, second, or third set. In this measurement, one of the RA(40) instances is scheduled across sockets, which is assumed to be suboptimal, as RA(40) is significantly more sensitive to NUMA placement than RA(1000). At the same time, the results show that the RA(40) instances nevertheless terminate earlier than the RA(1000) instance, despite one RA(40) instance being distributed across sockets while the RA(1000) instance remains socket-local.

Figure 5.12 illustrates the scheduling behavior of HARP with NUMA awareness enabled. Here, the two RA(40) instances are scheduled socket-local, while the threads of the RA(1000) instance are distributed across the remaining cores from both sockets. This placement strategy is intended to reduce the performance impact on the more NUMA-sensitive RA(40) programs, while assigning the less sensitive RA(1000) instance to the cross-socket configuration.

12-Apps scenario The scheduling behavior of EEVDF under the 12-Apps scenario is shown in Figure 5.13. Similar to the preceding scenario, a notable observation is that the RA(40) instances are only visible up to approximately 52 s, after which exclusively

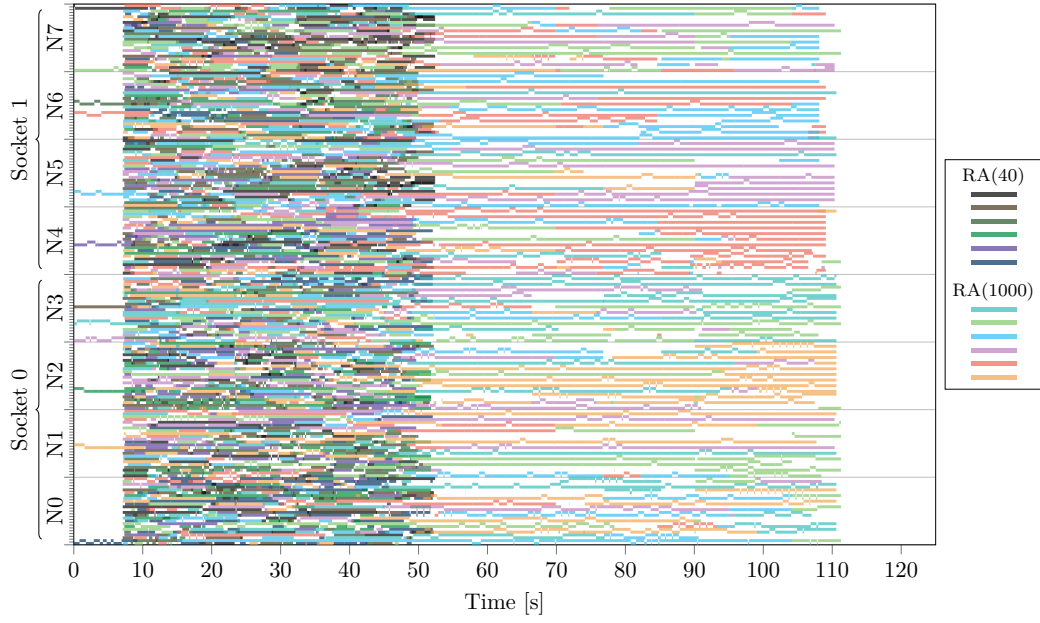


Figure 5.13: Thread-to-core assignments for the 12-Apps scenario managed by EEVDF

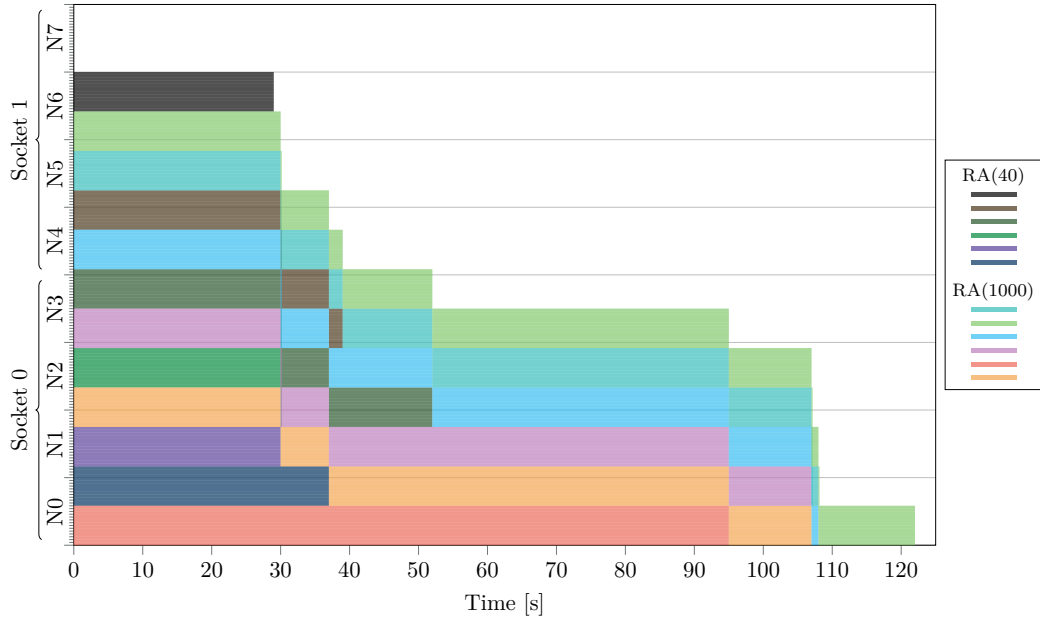


Figure 5.14: Thread-to-core assignments for the 12-Apps scenario managed by HARP without NUMA awareness

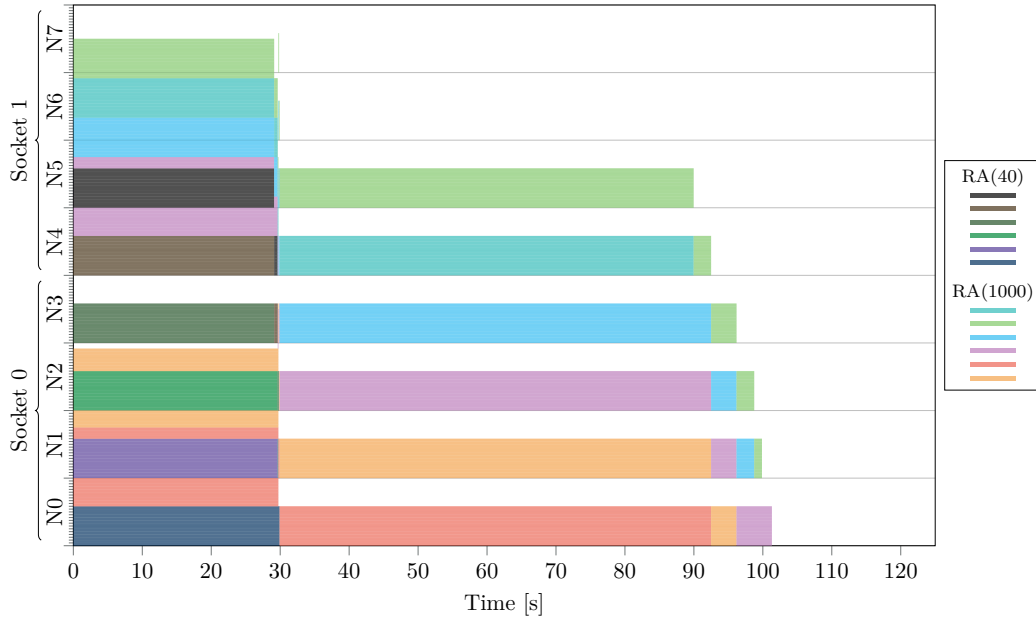


Figure 5.15: Thread-to-core assignments for the 12-Apps scenario managed by HARP with NUMA awareness

RA(1000) instances remain scheduled. Except for the yellow and orange threads, which towards the end primarily execute on nodes 2 and 4 respectively, no other clear placement patterns can be observed.

Figure 5.14 illustrates the scheduling behavior of HARP without NUMA awareness. Similar to the previous scenarios, HARP exhibits the expected naive allocation behavior. This results in one instance initially scheduled across sockets, while five instances are initially distributed across NUMA nodes but remain socket-local. Similar to the previous scenario, the assignment of each set of 14 cores to an application is determined solely by the order in which the applications register, making it effectively arbitrary which instance receives which cores.

Figure 5.15 shows the scheduling behavior of HARP with NUMA awareness enabled. In this case, HARP systematically allocates each RA(40) instance to a single NUMA node, while the RA(1000) instances are distributed at least socket-locally, so that no instance runs across sockets. However, there remains room for improvement. The six RA(40) instances are separated across six NUMA nodes, which would allow two RA(1000) instances to be scheduled NUMA-local as well. Since all RA(1000) instances are considered to have the same sensitivity to NUMA placement, the current resource allocation algorithm enforces socket-local placement for all RA(1000) instances, rather than attempting to schedule a subset of instances NUMA-local as long as feasible. For future work, it would be worthwhile to adapt the algorithm to exploit available NUMA-locality more effectively, even in cases where better locality is not explicitly required.

5.2.3.2 Performance impact

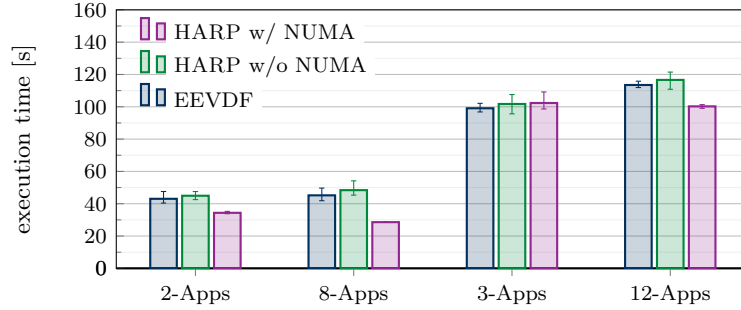


Figure 5.16: Absolute execution time for each scenario scheduled with EEVDF, NUMA-unaware HARP, and NUMA-aware HARP

After analyzing the placement decisions of the NUMA-aware version of HARP in comparison to EEVDF and the NUMA-unaware version of HARP, I now turn to the question of how these placement strategies affect overall execution time. The corresponding results are presented in Figure 5.16, which summarizes the four application scenarios discussed earlier. For each scenario, the figure reports the total execution time under EEVDF, NUMA-unaware HARP, and NUMA-aware HARP. The bars represent the average values obtained from 10 independent runs, and error bars indicate the observed variability.

In all but one scenario, NUMA-aware HARP achieves lower execution times than EEVDF, with an average improvement of up to 37 % and a geometric mean improvement of 15.5 %. This observation is consistent with the general assumption and key motivation behind introducing NUMA awareness into HARP, namely the reduction of costly remote memory accesses, which should in turn improve application performance.

The exception is the 3-Apps scenario, where the NUMA-aware version yields the longest execution times. As discussed in the placement analysis, this behavior can be explained by the fact that NUMA-aware HARP scheduled the longer running RA(1000) instance cross-socket while distributing the shorter RA(40) instances socket-local. Although this placement is optimal from a NUMA perspective, the RA(1000) instance generally dominates execution time. Thus, favoring RA(1000) for cross-socket scheduling inadvertently degrades total execution time in this particular case.

6 Conclusion and Future Work

6.1 Conclusion

In this work, I addressed two primary objectives. The first focused on integrating the scale-up database management system DuckDB into the HARP resource management framework, as detailed in Chapter 3. The second objective aimed to extend HARP to incorporate NUMA topology information into its scheduling decisions, which is discussed in Chapter 4.

The integration of DuckDB into HARP began with an analysis of DuckDB program structures to identify suitable entry points that would allow adjusting the database’s internal parallelism in response to HARP’s resource allocations. I identified the **TaskScheduler** class, responsible for managing task distribution and thread execution, as an appropriate mechanism for this purpose. However, its adaptability is limited by the fact that adjustments can only be applied at query boundaries, which imposes restrictions on effective resource allocation. In parallel, I investigated methods for extracting database-specific performance metrics for guiding HARP’s resource allocation. For this, DuckDB’s internal **ProgressBar** class, which tracks the number of rows processed during query execution, proved to be a suitable solution.

When integrating DuckDB into HARP, the evaluation of executing single DuckDB benchmarks revealed that, when pre-generated operating points were provided, HARP incurred a geometric mean degradation of 1.7 % in execution time and 0.4 % in energy consumption compared to CFS. While the overall geometric mean indicates a slight drawback, individual single-query scenarios achieved improvements of up to 9 % in performance and 57 % in energy efficiency. In contrast, scenarios involving multiple applications showed more consistent benefits, with a geometric mean improvement of 21 % in execution time and 28 % in energy consumption, and peak improvements of 67 % in performance and 73 % in energy efficiency.

When relying on online measurements without pre-generated operating points, the evaluation of single DuckDB programs revealed a more pronounced geometric mean degradation of 61 % in execution time and 10 % in energy consumption, primarily due to the limitation that DuckDB’s internal scaling adaptation occurs only at query boundaries. Nevertheless, individual scenarios showed improvements of up to 10 % in performance and 55 % in energy consumption. For scenarios involving multiple applications, the geometric mean indicated a 19 % degradation in performance alongside a 21 % improvement in energy consumption, with peak results reaching up to 67 % better performance and 21 % higher energy efficiency.

Extending HARP with NUMA-awareness began by making relevant system-level NUMA information accessible to the HARP resource manager. Then, I illustrated how HARP’s resource allocation algorithm can be adapted to incorporate this information. Determin-

ing from the server side, which applications benefit most from NUMA-aware scheduling, proved challenging. Therefore, in the current implementation, this information is provided by the applications themselves. Additionally, I introduced a **MigratePages** feature, allowing applications that are migrated between NUMA nodes to relocate their allocated data and reestablish memory locality.

Evaluation of the NUMA-aware extension demonstrates that application placement according to NUMA topology works as intended when HARP is provided with a given operating point. This NUMA-aware version of HARP achieves a geometric mean performance improvement of 15.5 % and peaks at 37 % compared to the EEVDF scheduler, highlighting the critical role of topology-aware scheduling in optimizing system performance.

6.2 Future Work

The results of the DuckDB integration into HARP show that the database’s coarse-grained scaling adaptations represent a major limitation, particularly when HARP relies on online measurements. Consequently, a promising direction for future research is to investigate how DuckDB’s internal tasking and threading mechanisms could be adapted to support reconfiguration during query execution. At the same time, these limitations in scaling granularity do not fully explain why single DuckDB programs show no overall benefit from the integration even when not relying on online measurements. Therefore, further investigation into the underlying causes would also be valuable.

The NUMA awareness extension of HARP proved effective, but this enhancement is still in an early stage of development, leaving multiple avenues for future work. The current evaluation is restricted to placement decisions under fixed operating points. Extending the evaluation to less constrained scenarios, such as when relying on online measurements, is an important next step. Moreover, the integrated data migration functionality across NUMA nodes represents a key feature whose usability should be broadened to support a wider range of applications.

The evaluation further revealed that when applications request less NUMA-locality than is technically achievable, the extension does not attempt to leverage the additional potential for locality, even in cases where doing so would not conflict with other requirements. Thus, refining the scheduling algorithm to exploit available NUMA-locality more effectively, even in the absence of explicit application requests, would be beneficial. Finally, a broader challenge for HARP is the incorporation of mechanisms that can determine when thread migrations are genuinely necessary or beneficial. As discussed in Section 5.2.3.1, and illustrated for example in Figure 5.6, migrating an application to newly freed cores can cause cross-socket memory accesses and additional context-switching overhead without yielding performance gains. Incorporating such mechanisms would help to avoid counterproductive migrations, thereby reducing overhead and maintaining data locality.

Bibliography

- [1] Apostolos Kotsiolis. *Simultaneous Multithreading: Driving Performance and Efficiency on AMD EPYC CPUs*. <https://www.amd.com/en/blogs/2025/simultaneous-multithreading-driving-performance-a.html>. [Online; accessed 30-August-2025].
- [2] Apple. *Apple unleashes M1*. [Online; accessed 15-September-2025]. 2020. URL: <https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/>.
- [3] David Bailey, E. Barszcz, Barton J.T, Browning D.S, Carter R.L, Dagum D, Fatoohi R.A, Paul Frederickson, Lasinski T.A, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Weeratunga K. “The Nas Parallel Benchmarks.” In: *International Journal of High Performance Computing Applications* 5 (Sept. 1991), pp. 63–73. DOI: 10.1177/109434209100500306.
- [4] Blaise Barney. *POSIX Threads Programming*. <https://computing.llnl.gov/tutorials/pthreads/>. [Online; accessed 10-September-2025]. 2009.
- [5] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. “Scalable and robust latches for database systems.” In: *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 2020, pp. 1–8.
- [6] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [7] Mei-Ling Chiang, Chieh-Jui Yang, and Shu-Wei Tu. “Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in NUMA multi-core systems.” In: *Journal of Systems and Software* 121 (2016), pp. 72–87.
- [8] ClickHouse. *ClickBench*. <https://github.com/ClickHouse/ClickBench>. [Online; accessed 30-August-2025]. 2025.
- [9] Jonathan Corbet. “An EEVDF CPU scheduler for Linux.” In: <https://lwn.net/Articles/925371/> (2023).
- [10] Yanlei Diao, Dominik Horn, Andreas Kipf, Oleksandr Shchur, Ines Benito, Wenjian Dong, Davide Pagano, Pascal Pfeil, Vikram Nathan, Balakrishnan Narayanaswamy, et al. “Forecasting algorithms for intelligent resource scaling: An experimental analysis.” In: *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 2024, pp. 126–143.
- [11] Linux Kernel Documentation. *CFS Scheduler*. [Online; accessed 21-August-2025]. URL: <https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html>.
- [12] Linux Kernel Documentation. *EEVDF Scheduler*. [Online; accessed 21-August-2025]. URL: <https://www.kernel.org/doc/html/latest/scheduler/sched-eevdf.html>.

- [13] DuckDB. *Benchmark Guide*. <https://github.com/duckdb/duckdb/tree/main/benchmark>. [Online; accessed 30-August-2025]. 2025.
- [14] DuckDB. *C++ API*. <https://duckdb.org/docs/stable/clients/cpp>. [Online; accessed 12-September-2025].
- [15] DuckDB. *duckdb*. <https://github.com/duckdb/duckdb>. [Online; accessed 12-September-2025]. 2025.
- [16] DuckDB. *DuckDB is a fast database system*. [Online; accessed 7-September-2025]. URL: <https://duckdb.org/>.
- [17] DuckDB. *Why DuckDB*. [Online; accessed 7-September-2025]. URL: https://duckdb.org/why_duckdb.html.
- [18] Marshall L Fisher. “The Lagrangian relaxation method for solving integer programming problems.” In: *Management science* 27.1 (1981), pp. 1–18.
- [19] Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Gilles Muller, and Vivien Quéma. *Scheduling Applications on NUMA Multicore Platforms with an Asymmetric Interconnect Topology*. <https://www.fabiengaud.net/resources/RR-LIG-XXX.pdf>.
- [20] Peter Greenhalgh. “Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7.” In: [Online; accessed 10-September-2025]. 2011. URL: <https://www.eetimes.com/big-little-processing-with-arm-cortex-a15-cortex-a7/>.
- [21] Dongsheng He, Lijun Chen, Jinrong Liang, Chunmei Kang, and Jiaqing Bai. “NUMA-Aware Contention Scheduling on Multicore Systems.” In: *2021 16th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*. IEEE. 2021, pp. 452–457.
- [22] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. “A case against (most) context switches.” In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021, pp. 17–25.
- [23] Intel. *Technical Overview Of The Intel® Xeon® Scalable processor Max Series*. [Online; accessed 25-August-2025].
- [24] Intel. *What Is Performance Hybrid Architecture?* [Online; accessed 10-September-2025]. URL: <https://www.intel.com/content/www/us/en/support/articles/000091896/processors.html>.
- [25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks.” In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: <https://doi.org/10.1145/3065386>.
- [26] Wei Li, Jiachi Zhang, Ye Yin, Yan Li, Zhanyang Zhu, Wenchao Zhou, Liang Lin, and Feifei Li. “Flux: decoupled auto-scaling for heterogeneous query workload in Alibaba AnalyticDB.” In: *Companion of the 2024 international conference on management of data*. 2024, pp. 255–268.

- [27] Geunsik Lim and Sang-Bum Suh. “User-level memory scheduler for optimizing application performance in numa-based multicore systems.” In: *2014 IEEE 5th International Conference on Software Engineering and Service Science*. IEEE. 2014, pp. 240–243.
- [28] Linux. *migrate_pages(2) — Linux manual page*. [Online; accessed 16-September-2025]. URL: https://www.man7.org/linux/man-pages/man2/migrate_pages.2.html.
- [29] Linux. *move_pages(2) — Linux manual page*. [Online; accessed 16-September-2025]. URL: https://man7.org/linux/man-pages/man2/move_pages.2.html.
- [30] Linux. *numactl(8) — Linux manual page*. [Online; accessed 16-September-2025]. URL: <https://www.man7.org/linux/man-pages/man8/numactl.8.html>.
- [31] Microsoft. *Microsoft SQL Server*. [Online; accessed 7-September-2025]. URL: <https://www.microsoft.com/de-de/sql-server>.
- [32] Patrick Mochel and Mike Murphy. *sysfs - The filesystem for exporting kernel objects*. [Online; accessed 4-September-2025].
- [33] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. “Intelligent scaling in amazon redshift.” In: *Companion of the 2024 International Conference on Management of Data*. 2024, pp. 269–279.
- [34] Oracle. *Database*. [Online; accessed 7-September-2025]. URL: <https://www.oracle.com/database>.
- [35] Yeonsu Park, Byungchul Tak, and Wook-Shin Han. “QaaD (query-as-a-data): scalable execution of massive number of small queries in spark.” In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–26.
- [36] PostgreSQL. *PostgreSQL: The World’s Most Advanced Open Source Relational Database*. [Online; accessed 7-September-2025]. URL: <https://www.postgresql.org/>.
- [37] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. “Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores.” In: *Proceedings of the VLDB Endowment* 10.2 (2016), pp. 37–48.
- [38] Yeasir Rayhan and Walid G Aref. “Revisiting Page Migration for Main-Memory Database Systems.” In: *arXiv preprint arXiv:2503.17685* (2025).
- [39] Steven Rostedt. *ftrace - Function Tracer*. [Online; accessed 9-September-2025]. URL: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt>.
- [40] Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. “Lsched: A workload-aware learned query scheduler for analytical database systems.” In: *Proceedings of the 2022 International Conference on Management of Data*. 2022, pp. 1228–1242.
- [41] SAP. *Was ist SAP HANA?* [Online; accessed 7-September-2025]. URL: <https://www.sap.com/germany/products/data-cloud/hana/what-is-sap-hana.html>.

- [42] Kanoj Sarcar. *What is NUMA?* [Online; accessed 25-August-2025]. URL: <https://www.kernel.org/doc/html/v4.18/vm/numa.html>.
- [43] Shekhar Kapoor. *What are Multi-Die Systems?* <https://www.synopsys.com/blogs/chip-design/what-are-multi-die-systems.html>. [Online; accessed 25-August-2025]. 2023.
- [44] Till Smejkal. *Energy-Aware Resource Management in Heterogeneous Systems based on Hierarchical Models*. Unpublished.
- [45] Till Smejkal, Robert Khasanov, Jeronimo Castrillon, and Hermann Härtig. “HARP: Energy-Aware and Adaptive Management of Heterogeneous Processors.” In: *Proceedings of the 26th ACM/IFIP International Middleware Conference (Middleware ’25)*. Accepted. Nashville, TN, USA: ACM, Dec. 2025.
- [46] Snowflake. *Sample Data: TPC-H*. <https://docs.snowflake.com/en/user-guide/sample-data-tpch>. [Online; accessed 30-August-2025].
- [47] Portworx Team. *Scale Up vs Scale Out: What is the Difference?* [Online; accessed 7-September-2025]. URL: <https://portworx.com/blog/scale-up-vs-scale-out/>.
- [48] TPC. *TPC-H Version 2 and Version 3*. <https://www.tpc.org/tpch/>. [Online; accessed 28-August-2025].
- [49] Benjamin Wagner, André Kohn, and Thomas Neumann. “Self-tuning query scheduling for analytical workloads.” In: *Proceedings of the 2021 international conference on management of data*. 2021, pp. 1879–1891.
- [50] Wikipedia. *Prozessor* — *Wikipedia, die freie Enzyklopädie*. <https://de.wikipedia.org/w/index.php?title=Prozessor&oldid=258775245>. [Online; accessed 26-August-2025]. 2025.
- [51] Wikipedia contributors. *Log-log plot* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Log-log_plot&oldid=1296342856. [Online; accessed 26-August-2025]. 2025.
- [52] Wikipedia contributors. *Non-uniform memory access* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Non-uniform_memory_access&oldid=1305078675. [Online; accessed 25-August-2025]. 2025.
- [53] Ting Wu, Jingting He, Ying Qian, and Weichen Liu. “Optimizing the performance of in-memory file system by thread scheduling and file migration under NUMA multiprocessor systems.” In: *Journal of Systems Architecture* 159 (2025), p. 103344.

Appendix A

Additional Results

A.1 Full Table of Performance Loss Measurements

Application	Memory Heaviness (LLC misses / 10^3 Instr.)	Performance Loss in %	
		Intra-Socket	Cross-Socket
ep.C	0.01	-0.6	-0.1
DuckDB queries (SF10)			
native_lineitem	2	1.7	10.5
native_orders	2	1.63	10.1
parquet_orders	2	1.63	11.0
csv_orders	0.7	0.73	14.4
csv_lineitem	1.1	1.07	10.7
Random Access with x computings			
0	35	4.4	55.5
2	22	5.7	57.5
4	20	7.2	61.1
6	18	8.2	64.0
8	16	9.5	67.8
10	15	10.3	70.8
12	14	11.4	73.2
14	13	11.8	74.6
16	12	12.4	75.5
18	11	13.0	77.7
20	9	13.6	79.4
40	6.5	17.0	85.1
60	5	15.7	77.1
80	3.5	13.9	69.1
100	3	13.0	64.7
120	3	11.9	59.6
140	2.5	11.1	55.4
160	2	10.4	51.3
180	2	9.8	48.1
200	1.8	9.2	45.2
400	1.1	5.9	28.8
600	0.8	4.3	20.9
800	0.5	3.3	16.5
1000	0.4	2.7	13.6

Table A.1: Performance loss (%) of different programs depending on their memory heaviness, measured in LLC misses per 10^3 instructions

A.2 Complete Measurements of the Evaluation of Scaling DuckDB

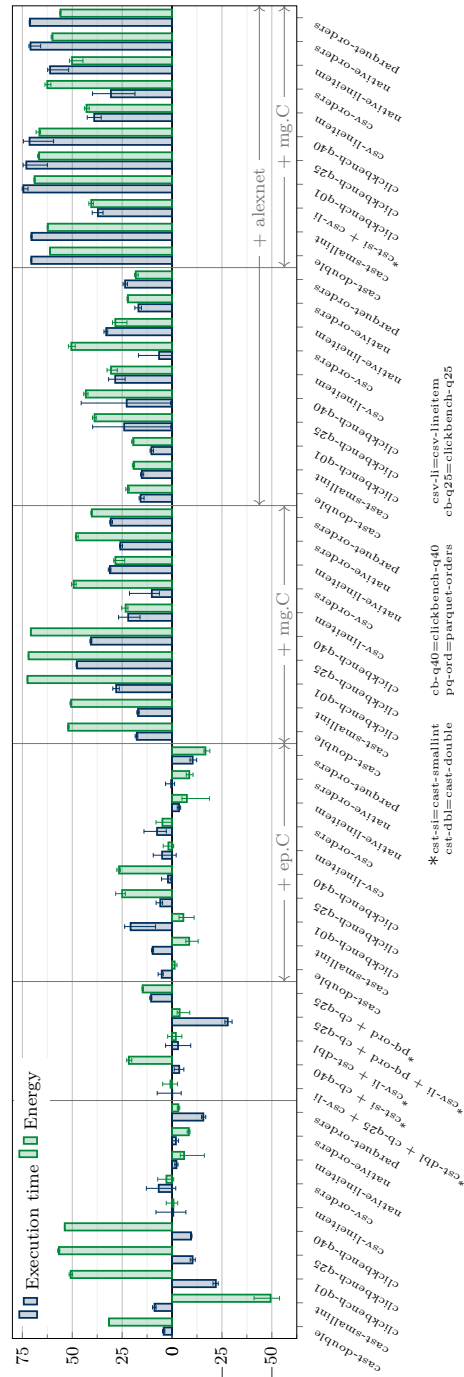


Figure A.1: Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and rows per second as performance metric.

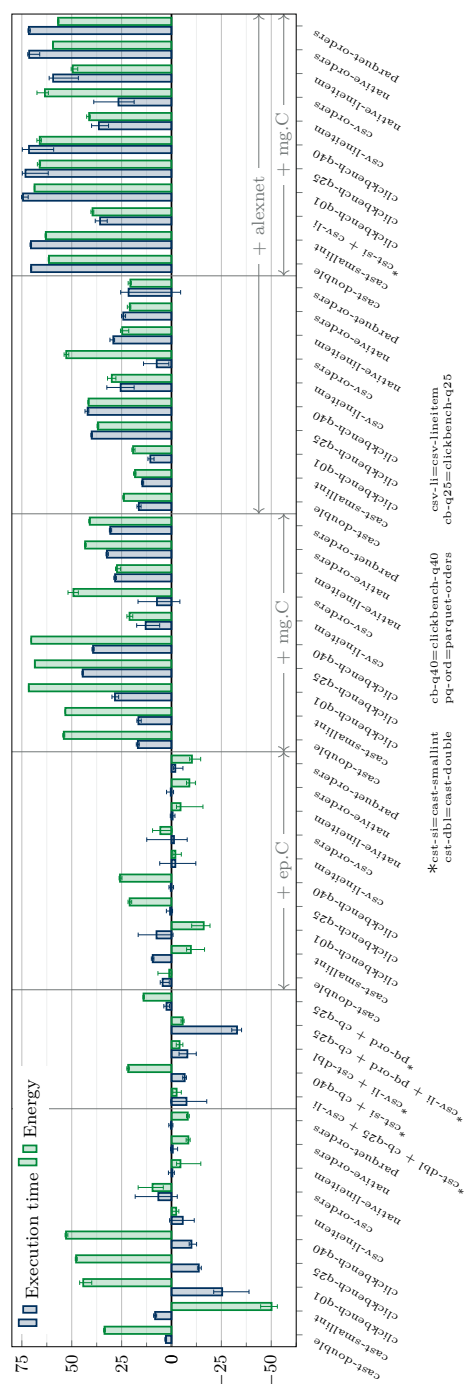
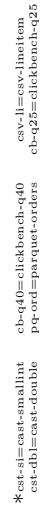


Figure A.2: Relative improvement in % of HARP over CFS. HARP runs with offline-generated operating points and instructions per second as performance metric.



generated operating points and rows per second as performance metric.